
PyMoDAQ Documentation

Release 4.2.0

Weber Sebastien

May 07, 2024

CONTENTS:

1	Training	3
2	Information	5
3	Credits	7
4	Contribution	9
5	They use it	11
6	Citation	13
7	Changelog	15
7.1	PyMoDAQ's overview	15
7.2	What's new in PyMoDAQ 4	17
7.2.1	Package hierarchy	17
7.2.2	Data Management	20
7.2.3	DAQ_Scan	20
7.3	User's Guide	20
7.3.1	Installation	20
7.3.2	How to Start	26
7.3.3	Configuration	27
7.3.4	DashBoard and Control Modules	30
7.3.5	Extensions	45
7.3.6	Data Management	93
7.3.7	Useful Modules	124
7.3.8	TCP/IP communication	133
7.3.9	LECO communication	139
7.4	Developer's Guide	140
7.4.1	Contributing	140
7.4.2	Plugins	144
7.4.3	Custom App	158
7.4.4	Managers and Mixin Objects	161
7.5	Tutorials	161
7.5.1	Git/GitHub	161
7.5.2	How to modify existing PyMoDAQ's code?	195
7.5.3	How to create a new plugin/package for PyMoDAQ?	207
7.5.4	Story of an instrument plugin development	213
7.5.5	How to contribute to PyMoDAQ's documentation?	239
7.5.6	Updating your instrument plugin for PyMoDAQ 4	245
7.5.7	Tutorial On Data Manipulation and analysis	246

7.6	Who use it?	260
7.6.1	Institutions using PyMoDAQ	260
7.6.2	What they think of PyMoDAQ?	260
7.6.3	Some Scientific publication on/using PyMoDAQ	261
7.7	Glossary Terms	261
7.8	Library Reference	263
7.8.1	Control modules	263
7.8.2	Extensions	268
7.8.3	Utility Modules	275
7.8.4	Utility Libraries	311
8	Indices and tables	347
	Python Module Index	349
	Index	351

PyMoDAQ is an open-source software, officially supported by the CNRS, to perform modular data acquisition with Python. It proposes a set of modules used to interface any kind of experiments. It simplifies the interaction with detector and actuator hardware to go straight to the data acquisition of interest.

French version [here](#)

TRAINING

PyMoDAQ
Modular Data Acquisition with Python

Actions 2024

Action Nationale de Formation
Développer PyMoDAQ
Les 17/18/19 juin 2024
à Toulouse
(laboratoire CEMES)

Journées PyMoDAQ
Réalizations, tables rondes et évolutions du logiciel
Du 21 au 23 octobre 2024
à Lyon
(Campus Claude Bernard)

Fig. 1.1: Training sessions announcement and PyMoDAQ's days

Note:

- Third edition of the PyMoDAQ's Days: Lyon 20/22 October 2024. Register on <https://pymodaq-jt2022.sciencesconf.org/>
- Training session in Toulouse, France 17/19 June 2024

PyMoDAQ has two purposes:

- First, to provide a complete interface to perform automated measurements or logging data without having to write a user/interface for each new experiment.
- Second, to provide various tools (User interfaces, classes dedicated to specific tasks...) to easily build a *Custom App*

It is divided in two main components as shown on figure Fig. 1.2

- The *Dashboard* and its control modules: *DAQ Move* and *DAQ Viewer*
- Extensions such as the *DAQ Scan* or the *DAQ Logger*

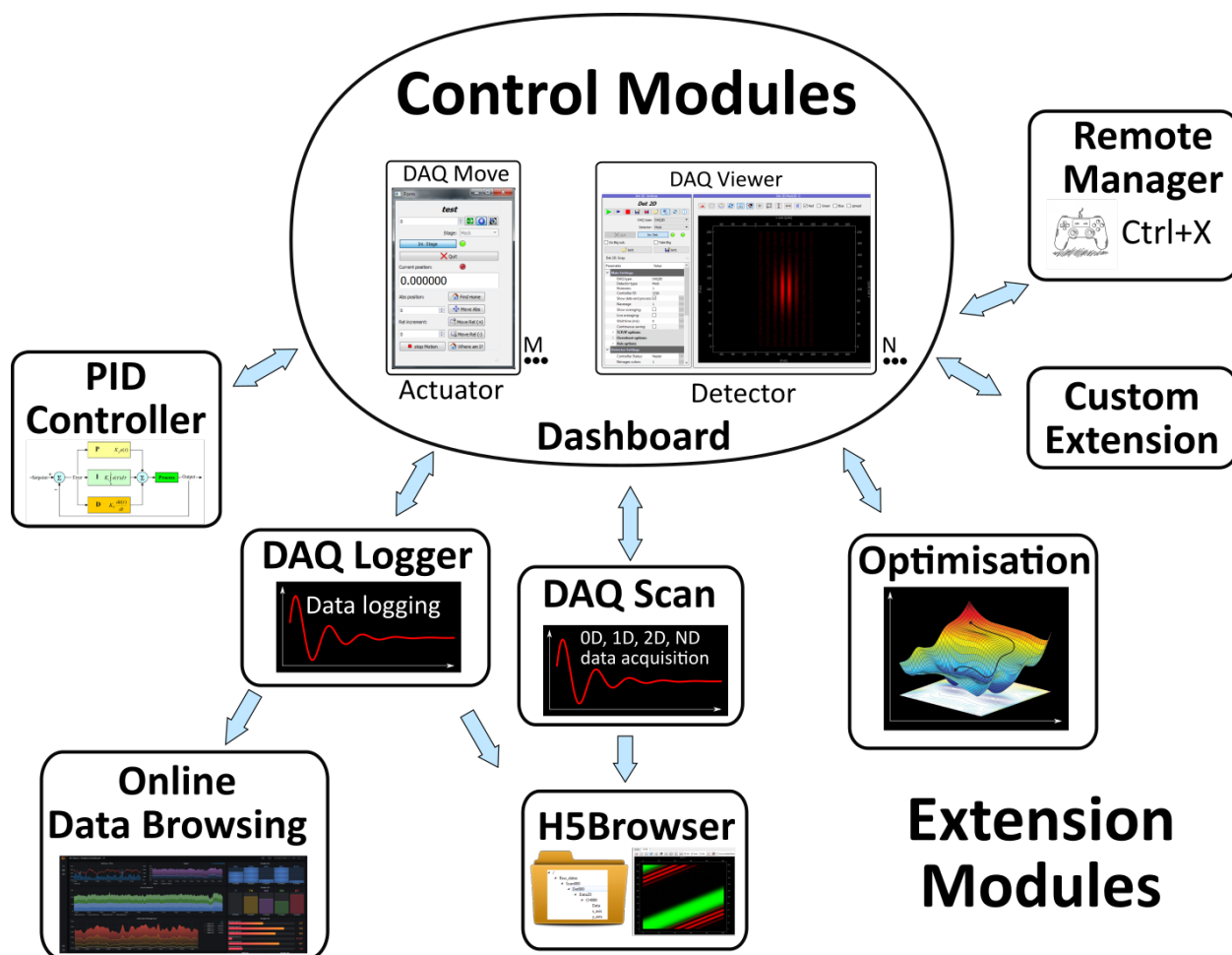


Fig. 1.2: PyMoDAQ's Dashboard and its extensions: *DAQ_Scan* for automated acquisitions, *DAQ_Logger* for data logging and many other.

The Control modules are interfacing real instruments using user written plugins. The complete list of available plugins is maintained on this [GitHub repository](#) and installed using the *Plugin Manager*

INFORMATION

GitHub repo: <https://github.com/PyMoDAQ>

Documentation: <http://pymodaq.cnrs.fr/>

Scientific [article](#) on Review of Scientific Instruments journal

General public [article](#) on Scientia

List of available [plugins](#)

Video tutorials [here](#)

Mailing List: <https://listes.services.cnrs.fr/www/info/pymodaq>

CREDITS

Based on the `pyqtgraph` library : <http://www.pyqtgraph.org> by Luke Campagnola.

PyMoDAQ is written by Sébastien Weber: sebastien.weber@cemes.fr under a MIT license.

CONTRIBUTION

If you want to contribute see this page: [Contributing](#)

THEY USE IT

See *Who use it?*

CITATION

By using PyMoDAQ, you are being asked to cite the article published in Review of Scientific Instruments [RSI 92, 045104 \(2021\)](#) when publishing results obtained with the help of its interface. In that way, you're also helping in its promotion and amelioration.

CHANGELOG

Please see the changelog.

7.1 PyMoDAQ's overview

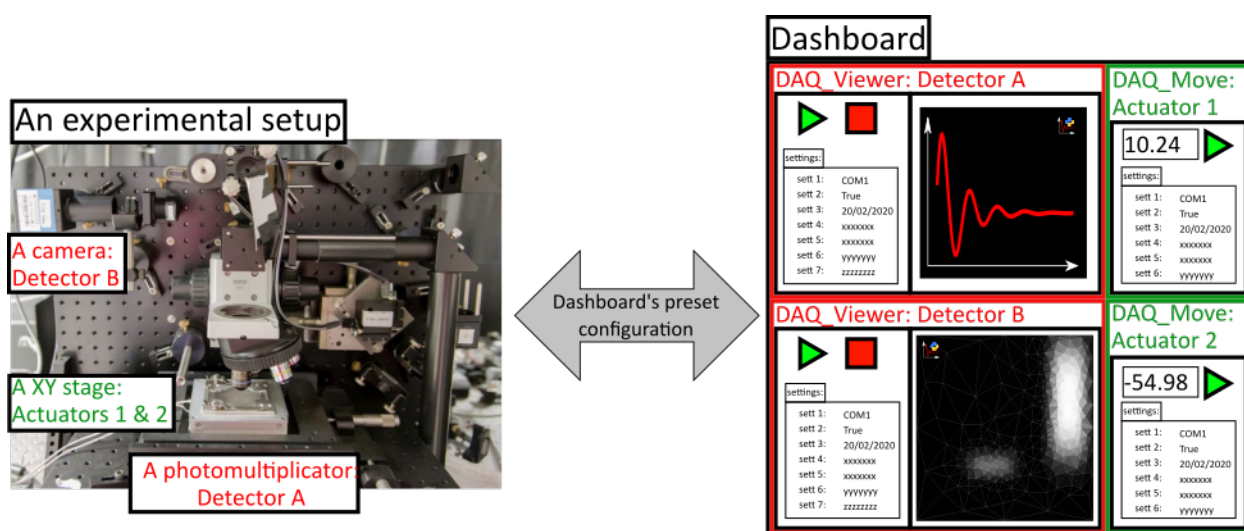


Fig. 7.1: PyMoDAQ control of an experimental setup using the Dashboard and a set of DAQ_Vviewer and DAQ_Move modules

PyMoDAQ is an advanced user interface to control instruments (casually called Detectors) and actuators (sometimes called Moves for historical reasons). Each of these will have their own interface called *DAQ Viewer* and *DAQ Move* that are always the same (only some specifics about communication with the controller will differ), so that a PyMoDAQ's user will always find a known environment independent of the kind of instruments it controls. These detectors and actuators are grouped together in the *Dashboard* and can then be controlled manually by the user: acquisition of images, spectra... for various positions of the actuators (see Fig. 7.1). The Dashboard has functionalities to fully configure all its detectors and actuators and save the configuration in a file that will, at startup, load and initialize all modules. Then Dashboard's extensions can be used to perform advanced and automated tasks on the detectors and actuators (see Fig. 7.2):

- The first of these extensions is called *DAQ Scan* and is used to perform automated and synchronized data acquisition as a function of multiple actuators *positions*. Many kind of *scans* are possible: 1Ds, 2Ds, NDs, set of points and many ways to perform each of these among which *Adaptive* scan modes have been recently developed (from version 2.0.1).

- The second one is the *DAQ Logger*. It is a layer between all the detectors within the dashboard and various ways to log data acquired from these detectors. As of now, one can log to :
 - a local binary hdf5 file
 - a distant binary hdf5 file or *same as hdf5* but on the cloud (see [HSDS from the HDF group](#) and the [h5pyd](#) package)
 - a local or distant SQL Database (such as PostgreSQL). The current advantage of this solution is to be able to access your data on the database from a web application such as [Grafana](#). Soon a tutorial on this!!
- Joystick control of the dashboard actuators (and eventually detectors).
- PID closed loop interface
- Direct code execution in a Console

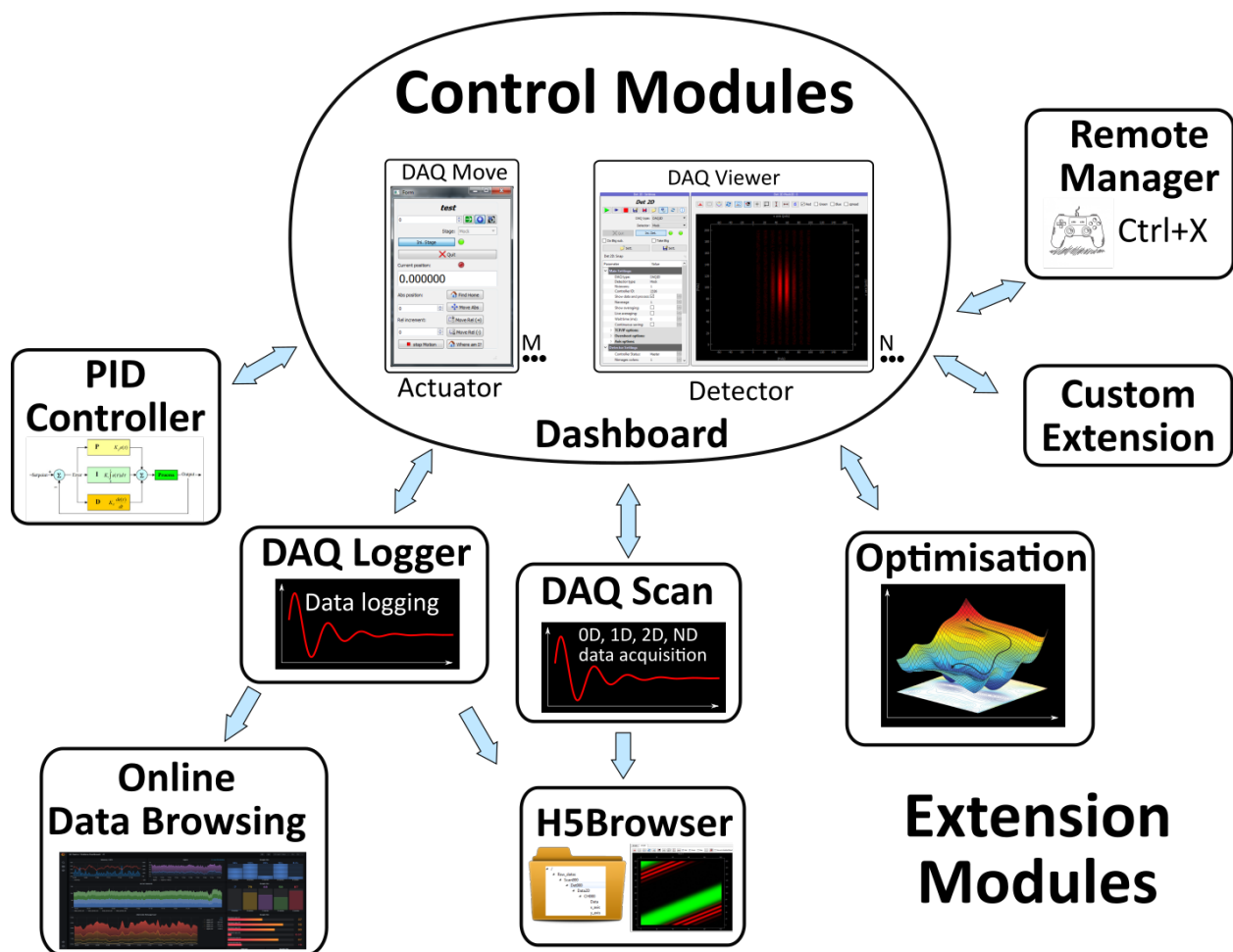


Fig. 7.2: PyMoDAQ's Dashboard and its extensions: DAQ_Scan for automated acquisitions, DAQ_Logger for data logging and many other.

7.2 What's new in PyMoDAQ 4

The main modifications in PyMoDAQ 4 is related to the hierarchy of the *modules* in the source code and the *data management*.

The feel and shape of the control modules and the way the DAQ_Scan work have been reworked. A new extension is introduced: the Console.

7.2.1 Package hierarchy

Before many modules where stored in a generic `daq_utils` module. It was kind of messy and the development of much nicer code for pymo4 was the occasion to reshape the package and its modules. Figure Fig. 7.3 shows the new layout of the package.

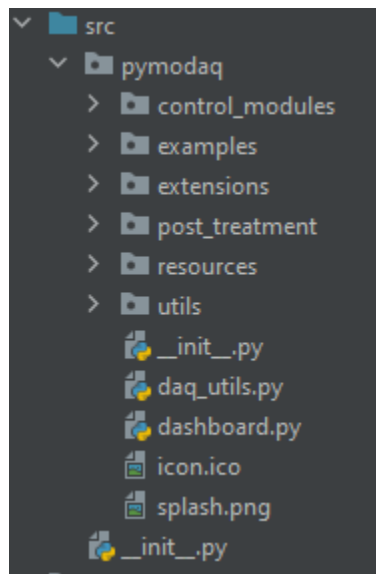


Fig. 7.3: Layout of the PyMoDAQ 4 package.

The only python file at the root is the `dashboard.py` that contains the code about the dashboard, the starting point of PyMoDAQ usage.

Note: There is a `daq_utils.py` file here as well to provide some back compatibility with pymodaq v3 but this file will soon be deprecated (when all plugins will be updated according to this tutorial)

Then you'll find modules for:

- *Control modules*: the DAQ_Viewer and the DAQ_Move and their utility modules
- Example module: contains some executable code to illustrate some features
- *Extension module*: contains the main extension of the DashBoard: DAQ_Scan, DAQ_Logger, PID and H5Browser
- Post-Treatment modules: utilities to process PyMoDAQ's data
- Resources module: contains the UI icons, templates for configuration and presets
- Utils module: contains all utility modules, see Fig. 7.4.

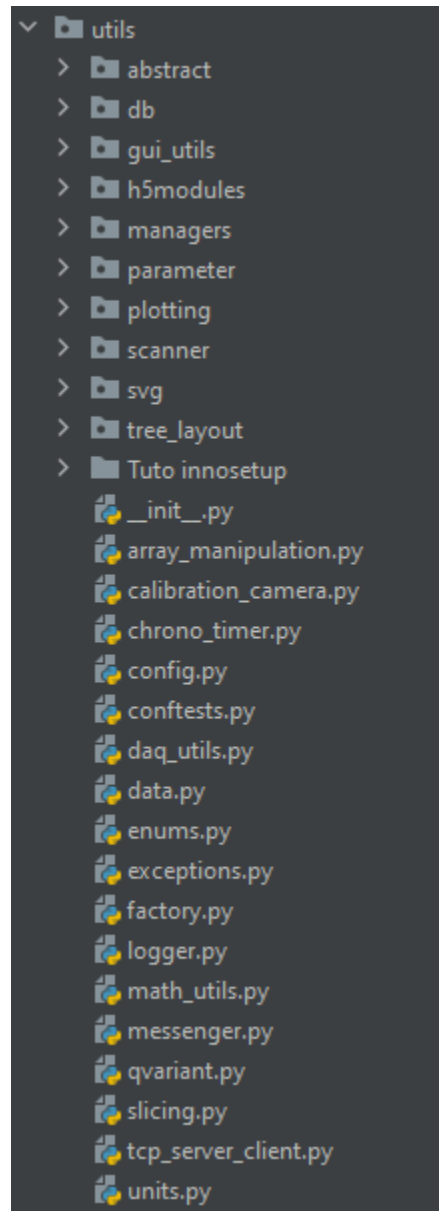


Fig. 7.4: Layout of the utils module

This last `utils` module contains many other module needed for PyMoDAQ to run smoothly. They can also be used in some other programs to use their features. Below is a short description of what they are related to:

- `abstract`: contains abstract classes (if not stored in another specific module)
- `db`: module related to data logging towards database (postgresql for instance)
- `gui_utils`: usefull UI widgets and related objects to build quickly and nicely user interfaces
- `h5modules`: everything related to the saving and browsing of data in hdf5 files
- `managers`: integrated objects managing various thing, for instance, control modules, presets, roi... In general they have a specific UI (that you can incorporate in your main UI) and the code to interact with whatever is related to it.
- `parameter`: extensions of the pyqtgraph Parameter introducing other widgets and Parameter types. Includes also serializers from/to Parameter to/from XML
- `plotting`: everything related to the plotting of data: including the 4 main data viewers, see [Plotting Data](#)
- `scanner`: objects related to the DAQ_Scan defining and managing the scans. The different types of scans are defined using a factory pattern.
- `svg`: under tests to plot svg
- `array_manipulation`: utility functions to create, manipulate and extract info from numpy arrays
- `calibration_camera`: utility UI to get a calibration file from a Camera compatible with pymodaq (to use real physical axes and not pixels in the data viewers). Old code, maybe to update for it to work
- `chrono_timer`: user interface to be used for timing things, see [ChronoTimer](#)
- `config`: objects dealing with configuration files (for instance the main config for pymodaq). Can be used elsewhere, for instance in instrument plugin
- `conftests`: configuration file for the test suite
- `daq_utils`: deprecated
- `data`: module containing all objects related to [Data Management](#)
- `enums`: base class and method to ease the use of enumerated types
- `exceptions`: contains some shared exceptions. But exceptions should be in their related module...
- `factory`: base class to be used when defining a factory pattern
- `logger`: methods to initialize the logging objects in the various modules
- `math_utils`: a set of useful mathematical functions
- `messenger`: function to be used when one want to display messages (in the log or in popups)
- `qvariant`: definition of a QVariant object. To be used in PySide as it is not defined there...
- `slicing`: definition of slicing objects used in the data management to slice data
- `tcp_server_client`: set of classes to build TCP/IP communication
- `units`: methods for conversion between physical units (especially photon energy in eV, nm, cm, J...)

7.2.2 Data Management

See *data management*.

7.2.3 DAQ_Scan

See *DAQ Scan*.

7.3 User's Guide

7.3.1 Installation

- *Preamble*
- *Setting up a new environment*
- *Installing PyMoDAQ*
- *Creating shortcuts on Windows*
- *Plugin Manager*
- *What about the Hardware*

Preamble

PyMoDAQ is written in [Python](#) and uses Python 3.7+. It uses the [Qt5](#) library (and a python Qt5 backend, see [Qt5 backend](#)) and the excellent [pyqtgraph](#) package for its user interface. For PyMoDAQ to run smoothly, you need a Python distribution to be installed. Here are some advices.

On all platforms **Windows**, **MacOS** or **Linux**, [Anaconda](#) or [Miniconda](#) is the advised distribution/package manager. Environments can be created to deal with different version of packages and isolate the code from other programs. Anaconda comes with a full set of installed scientific python packages while *Miniconda* is a very light package manager.

Setting up a new environment

- Download and install Miniconda3.
- Open a console, and cd to the location of the *condabin* folder, for instance: `C:\Miniconda3\condabin`
- Create a new environment: `conda create -n my_env python=3.8`, where *my_env* is your new environment name, could be *pymodaq353* if you plan to install PyMoDAQ version 3.5.3 for instance.. This will create the environment with python version 3.8 that is currently the recommended one, see [Python Versions](#).
- Activate your environment so that only packages installed within this environment will be *seen* by Python: `conda activate my_env`

Installing PyMoDAQ

Easiest part: in your newly created and activated environment enter: `pip install pymodaq`. This will install the latest PyMoDAQ available version and all its dependencies. For a specific version enter: `pip install pymodaq==x.y.z`.

Qt5 backend

PyMoDAQ source code uses a python package called `qtpy` that add an abstraction layer between PyMoDAQ's code and the actual Qt5 python implementation (either PyQt5 or PySide2, and soon PyQt6 and PySide6). `Qtpy` will look on what is installed on your environment and load PyQt5 by default (see the [PyMoDAQ configuration for default values](#) to change this default behaviour). This means you have to install one of these backends on your environment using either:

- `pip install pyqt5`
- `pip install pyside2` (still some issues with some parts of pymodaq's code. If you want to help fix them, please, don't be shy!)
- `pip install pyqt6` (not tested yet)
- `pip install pyside6` (not tested yet)

Linux installation

For Linux installation, only Ubuntu operating system are currently being tested. In particular, one needs to make sure that the QT environment can be used. Running the following command should be sufficient to start with:

```
sudo apt install libxkbcommon-x11-0 libxcb-icccm4 libxcb-image0 libxcb-keysyms1
libxcb-randr0 libxcb-render-util0 libxcb-xinerama0 libxcb-xfixes0 x11-utils
```

It is also necessary to give some reading and writing permission access to some specific folders. In particular, PyMoDAQ creates two folders that are used to store configurations files, one assigned to the system in `/etc/pymodaq/` and one assigned to the user `~/pymodaq/`. We need to give reading/writing permission access to the system folder. One should then run before/after installing pymodaq:

- `sudo mkdir /etc/pymodaq/`
- `sudo chmod uo+rw /etc/pymodaq`

As a side note, these files are shared between different pymodaq's versions (going from 3 to 4 for example). It is suggested to delete/remake the folder (or empty its content) when setting up a new environment with a different pymodaq version.

Creating shortcuts on Windows

Python packages can easily be started from the command line (see [How to Start](#)). However, Windows users will probably prefer using shortcuts on the desktop. Here is how to do it (Thanks to Christophe Halgand for the procedure):

- First create a shortcut (see [Fig. 7.5](#)) on your desktop (pointing to any file or program, it doesn't matter)
- Right click on it and open its properties (see [Fig. 7.6](#))
- On the *Start in* field ("Démarrer dans" in french and in the figure), enter the path to the `condabin` folder of your miniconda or anaconda distribution, for instance: `C:\Miniconda3\condabin`

- On the *Target* field, (“Cible” in french and in the figure), enter this string: `C:\Windows\System32\cmd.exe /k conda activate my_env & python -m pymodaq.dashboard`. This means that your shortcut will open the windows’s command line, then execute your environment activation (*conda activate my_env* bit), then finally execute and start **Python**, opening the correct pymodaq file (here *dashboard.py*, starting the Dashboard module, *python -m pymodaq.dashboard* bit)
- You’re done!
- Do it again for each PyMoDAQ’s module you want (to get the correct python file and it’s path, see [From command line tool](#):).

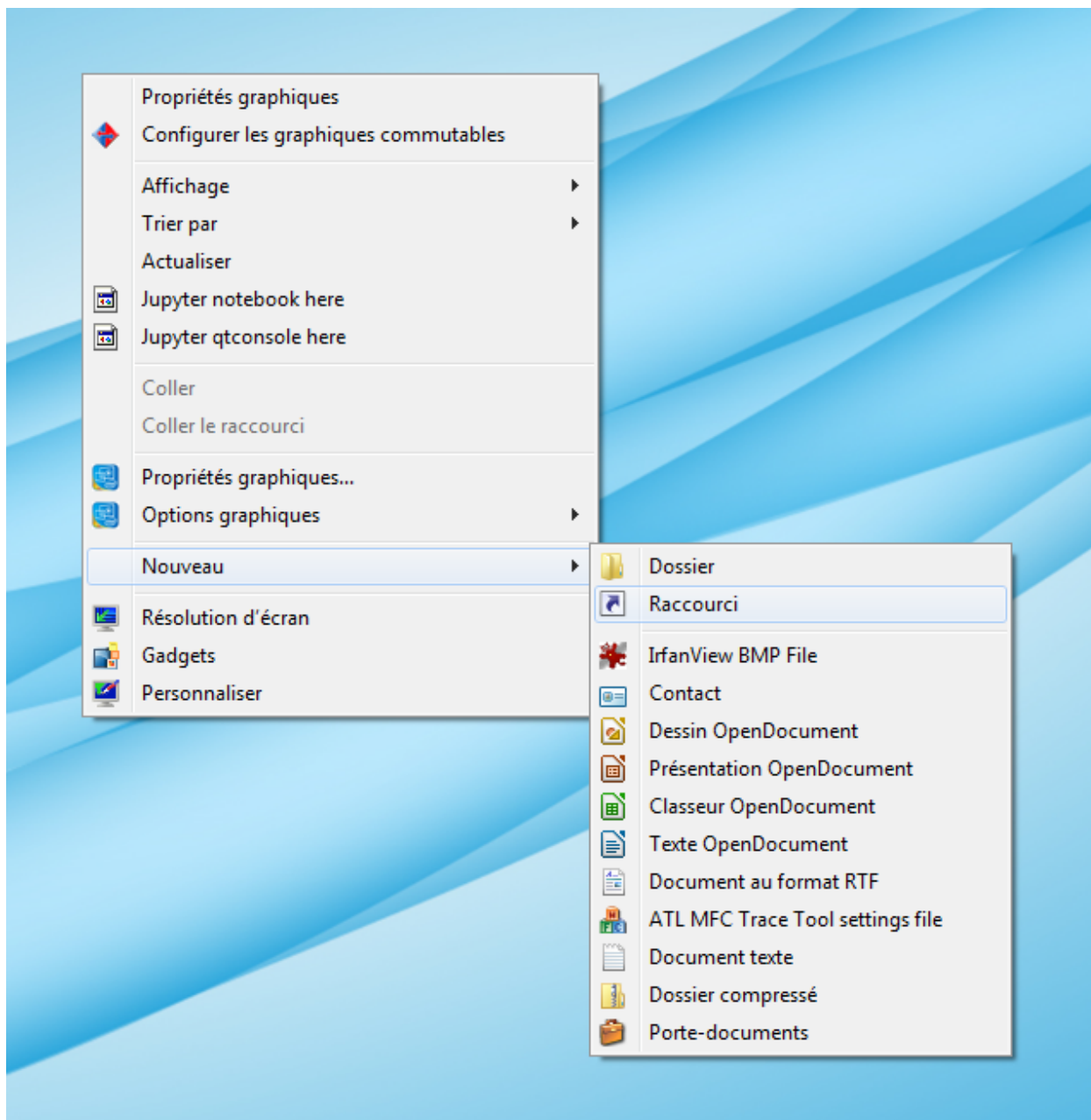


Fig. 7.5: Create a shortcut on your desktop

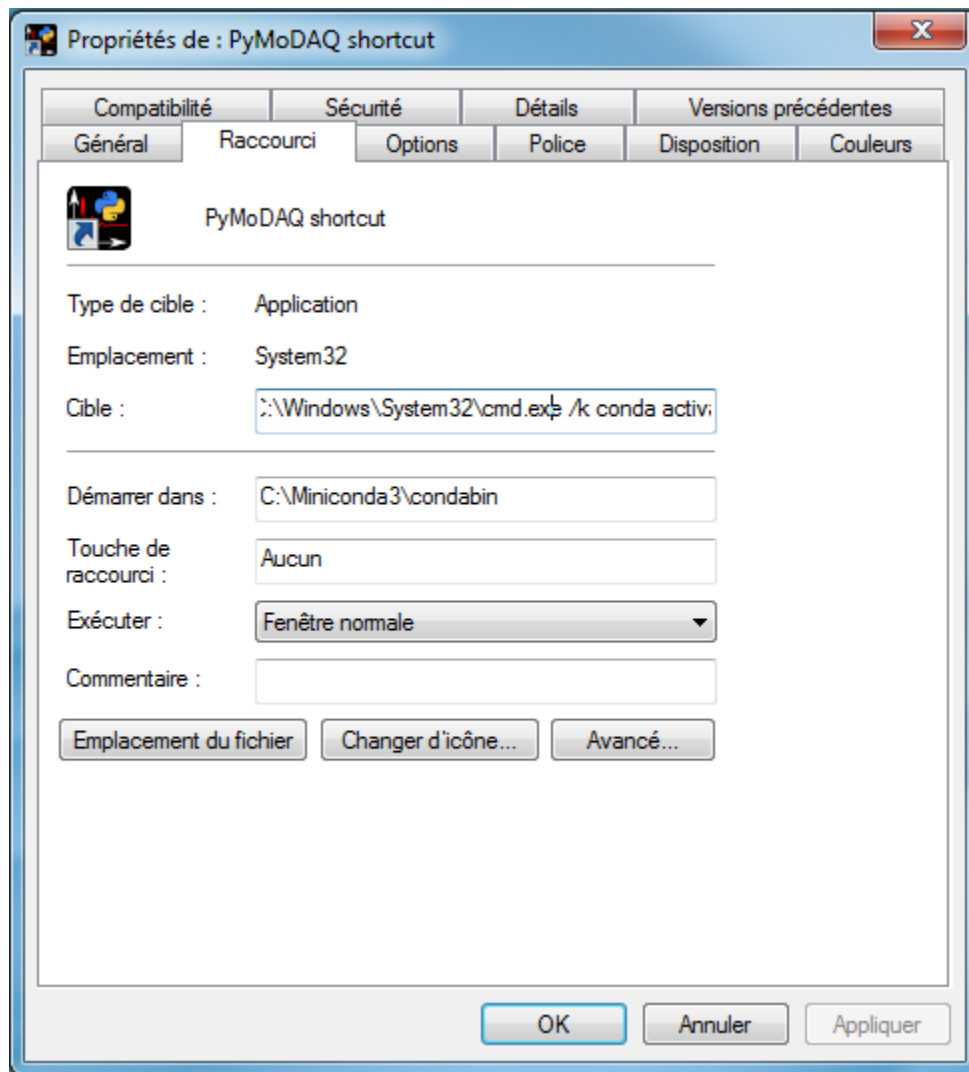


Fig. 7.6: Shortcut properties

Plugin Manager

Any new hardware has to be included in PyMoDAQ within a *plugin*. A PyMoDAQ's plugin is a python package containing several added functionalities such as instruments objects. A instrument object is a class inheriting from either a `DAQ_Move_Base` or a `DAQ_Viewer_Base` class and implements mandatory methods for easy and quick inclusion of the instrument within the PyMoDAQ control modules.

The complete list of available Instrument Plugins is maintained on this [GitHub repository](#).

While you can install them manually (for instance using `pip install plugin_name`), from PyMoDAQ 2.2.2 a plugin manager is available. You can open it from the **Dashboard** in the help section or directly using the command line: `python -m pymodaq_plugin_manager.manager` or directly `plugin_manager`

This will open the Plugin Manager User Interface as shown on figure [Fig. 7.7](#) listing the available plugins packages that can be either *installed*, *updated* or *removed*. It includes a description of the content of each package and the instruments it interfaces. For instance, on figure [Fig. 7.7](#), the selected *Andor* plugin package is selected and includes two plugins: a `Viewer1D` to interface Andor Shamrock spectrometers and a `Viewer2D` to interface Andor CCD camera.

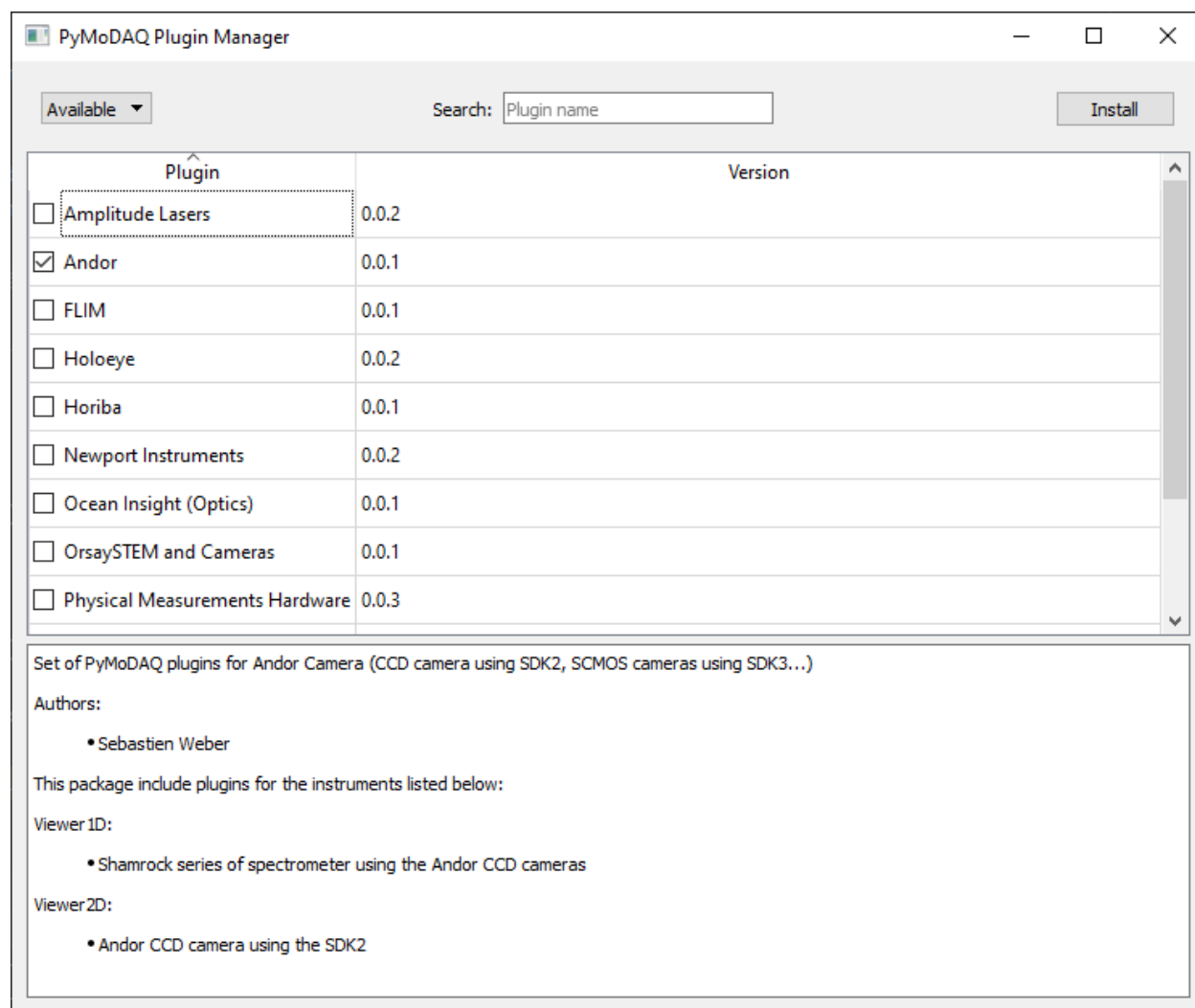


Fig. 7.7: Plugin Manager interface

What about the Hardware

So far, you've installed all the software layer managing Instrument control from the user up to the manufacturer driver. This means you still have to install properly your specific hardware. For this, there is no general recipe but below you'll find some advices/steps you can follow.

Serial/GPIB based hardware

In the case where your instrument is controlled using ASCII commands (basically strings), no more steps than plugging you instrument is needed. Just make sur the COM port or GPIB address is correct.

Library based hardware

In the case of instruments using a specific manufacturer driver (*.dll*, *.so* or *.NET* libraries) then you could follow these steps:

- Install the SDK/dll driver from the manufacturer
- Test the communication is fine using the software provided by the manufacturer (if available)
- Make sure your OS (Windows, Mac or linux) is able to find the installed library (if needed add the *path* pointing to your library in the **PATH** environment variable of your operating system
- Install the right PyMoDAQ's plugin
- You should be good to go!

Warning: From Python 3.8 onwards, the way python looks for dlls on your system changed causing issues on existing plugins using them. So far the right way was to add the path pointing to your dll in the system PATH environment variable. This no longer works and ctypes LoadLibrary function raises an error. A simple solution to this issue, is to add in the preamble of my/your plugins this instruction:

```
import os
os.add_dll_directory(path_dll)
```

where path_dll is the path pointing to your dll.

Note: Example: if you want to use a NI-DAQ instrument. You'll have to first install their driver Ni-DAQmx, then test you hardware using their MAX software and finally configure it using *pymodaq_plugins_daqmx* plugin.

Python Versions

As of today (early 2022), PyMoDAQ has been efficiently used on python 3.8 up to 3.9 versions. It's source code is regularly tested against those versions. Work is in progress to make it working with python 3.10/3.11, but some of PyMoDAQ's dependencies are not yet available for these versions.

7.3.2 How to Start

Various ways are possible in order to start modules from PyMoDAQ. In all cases after installation of the package (using pip or setup.py, see [Installation](#)) all the modules will be installed within the `site-packages` folder of python.

From command line tool:

Open a command line and **activate your environment** (if you're using anaconda, miniconda, venv...):

Load installed scripts

During its installation, a few scripts have been installed within you environment directory, this means you can start PyMoDAQ's main functionalities directly writing in your console either:

- `dashboard`
- `daq_scan`
- `daq_logger`
- `daq_viewer`
- `daq_move`
- `h5browser`
- `plugin_manager`

Execute a given python file

If you know where, within PyMoDAQ directories, is the python file you want to run you can enter for instance:

- `python -m pymodaq.dashboard`
- `python -m pymodaq.extensions.daq_scan`
- `python -m pymodaq.extensions.daq_logger`
- `python -m pymodaq.control_modules.daq_viewer`
- `python -m pymodaq.control_modules.daq_move`
- `python -m pymodaq.extensions.h5browser`
- `python -m pymodaq_plugin_manager.manager`

for PyMoDAQ's main modules. The `-m` option tells python to look within its *site-packages* folder (where you've just installed pymodaq) In fact if one of PyMoDAQ's file (`xxx.py`) as an entry point (a `if __name__=='__main__':` statement at the end of the file), you can run it by calling python over it...

Create windows's shortcuts:

See *Creating shortcuts on Windows* !

7.3.3 Configuration

All configuration files used by PyMoDAQ will be located within two folders each called *.pymodaq*. One is system wide and located in one of these locations:

- Windows: *ProgramData* folder
- Mac: *Library/Application Support* folder
- Linux: */etc*

while the other is restricted to the current user and located in the user's *home* folder.

All configuration files that should be shared between users are in the system wide folder, for instance all files related to the dashboard, see Fig. Fig. 7.8:

- preset configs: preset file defining the type and numbers of control modules for a given experiment
- batch configs: file describing the batch of scans to do
- layout configs: store the user interface docks arrangement
- overshoot configs: store the files defining overshoots
- remote configs: store the files defining the remote control actions
- roi configs: store the overall ROI in all *DAQ_Viewers* on the dashboard

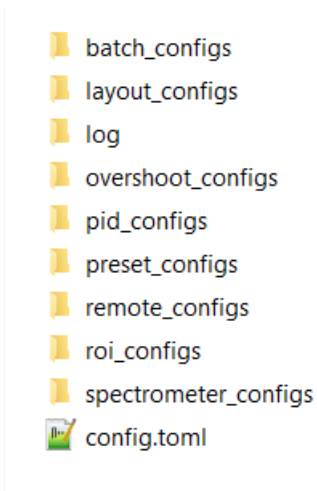


Fig. 7.8: Local folder to store configuration files

Configs from Managers

Each folder contains dedicated files: the log as text file and all module configuration files as xml files. These files are generated by dedicated managers when the user is configuring one aspect of PyMoDAQ, for instance using the *Preset manager* for defining Actuators and Detectors in the *Dashboard*. Apart the *log*, a user should not interact directly with those but use their respective manager user interface to create and modify them.

PyMoDAQ configuration for default values

The *config_pymodaq.toml* file is the only exception. It is there so that a particular user could enter specific personal information such as the name that will be used by default in the metadata, default *preset* file to load if executing directly the *DAQ_Scan* extension, default type of *Scan* and so on. The file can be directly modified but should be accessed within the *Dashboard* in the file menu.

The configuration file located in the system wide folder is the default one, see below, but when a user override the default values, they will be stored in another *config_pymodaq.toml* in the user *.pymodaq* folder. In this way, if the computer is shared among multiple users, each can specify their own metadata, UI feel and shape, default presets, ...

Below is a non exhaustive list of configuration entries stored in the *config_pymodaq.toml* file:

Listing 7.1: Default Configuration file of PyMoDAQ that will be copied on the local folder where the user can modify it

```
[data_saving]
  [data_saving.h5file]
    save_path = "C:\\\\Data" #base path where data are automatically saved
    compression_level = 5 # for hdf5 files between 0(min) and 9 (max)

    [data_saving.hsds] #hsds connection option (https://www.hdfgroup.org/solutions/
↳highly-scalable-data-service-hsds/)
    #to save data in pymodaq using hpyd backend towards distant server or cloud
↳(mimicking hdf5 files)
    root_url = "http://hsds.sebastienweber.fr"
    username = "pymodaq_user"
    pwd = "pymodaq"

[general]
debug_level = "DEBUG" #either "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"
debug_levels = ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]
check_version = true #automatically check version at startup

[user]
name = "User name" # default name used as author in the hdf5 saving files

[network]
  [network.logging]
    [network.logging.user]
      username = "pymodaq_user"
      pwd = "pymodaq"

    [network.logging.sql] #location of the postgresql database server and options
↳where the DAQ_Logger will log data
      ip = "10.47.3.22"
      port = 5432
```

(continues on next page)

(continued from previous page)

```
[network.tcp-server]
ip = "10.47.0.39"
port = 6341

[presets]
default_preset_for_scan = "preset_default"
default_preset_for_logger = "preset_default"
```

Plugins configuration for default values

In the same way, the file *config_pymodaq.toml* stores (system/user wide) default configuration values, plugins benefits of the same features. The mechanism is as follow. The plugin package should contain (PyMoDAQ >= 4) a resources folder containing at least the *VERSION* file and a *config_template.toml* file, see Fig. 7.9.

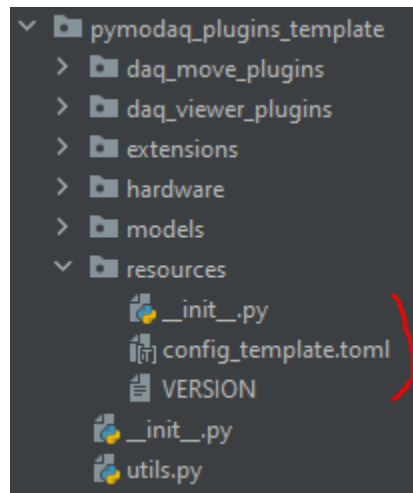


Fig. 7.9: Files in the resources folder of each plugin (well that should be like this as of october 2023)

This *config_template.toml* file holds any mandatory config values needed from within you plugin package scripts. The first time the plugin package is imported, this config will be copied into the system wide/user folders, to be used by the plugins scripts. They can be manually amended by each user in their *.pymodaq* user folder.

Another file is mandatory, the *utils.py* at the root of the plugin package, see Fig. 7.9. In there, will be defined the particular *Config* object to be used with each script of the package plugin:

```
class Config(BaseConfig):
    """Main class to deal with configuration values for this plugin"""
    config_template_path = Path(__file__).parent.joinpath('resources/config_template.toml')
    config_name = f"config_{__package__.split('pymodaq_plugins-')[1]}"
```

This object will automatically be linked to the system wide/user *.pymodaq* folder where the template will be copied and renamed from the plugin name. For instance, the plugin package, *pymodaq_plugins_optimisation* will produce a configuration file called *config_optimisation.toml*

7.3.4 DashBoard and Control Modules

DashBoard

This module is the heart of PyMoDAQ, it will:

- Help you declare the list of actuators and detectors to be used for a given experiment (*Preset manager*)
- Setup automatic data acquisition of detectors as a function of one or more actuators using its DAQ_Scan extension
- Log data into advanced binary file or distant database using its DAQ_Logger extension

The flow of this module is as follow:

- At startup you have to define/load/modify a preset (see *Preset manager*) representing an ensemble of actuators and detectors
- Define/load/modify eventual overshoots (see *Overshoot manager*)
- Define/load/modify eventual ROI (Region of interests) selections (see *ROI manager*)
- Use the actuators and detectors manually to drive your experiment
- Select an action to perform: automated scan (DAQ_Scan) and/or log data (DAQ_Logger)

Introduction

This module has one main window, the dashboard (Fig. 7.10) where a log and all declared actuators and detectors will be loaded as instances of DAQ_Move and DAQ_Viewer. The dashboard gives you full control for manual adjustments of each actuator, checking their impact on live data from the detectors. Once all is set, one can move on to different actions.

Menu Bar Description

Figure Fig. 7.11 displays the menu of the *Dashboard* window with access to all the *Managers* useful within PyMoDAQ and described below:

The **file** menu will allow you to quickly display, in a default text editor, the current log file (older logs can be found in the *pymodaq_local* folder, see *Configuration*). The user can also access and edit the general configuration file *config.toml* selecting the *Show configuration file* entry that will open a popup window (see Fig. Fig. 7.12) allowing the user to modify all its fields. Finally, the user can *Quit* the application or *Restart* it if changes have to be applied (for instance when modifying a *Preset*)

The **Settings** menu is allowing the user to save/load layouts of docked windows within the *Dashboard*.

Note: Docked Windows Layout: when a *Preset* has been loaded and if the arrangement of the *Control Modules* (their docked panels) is modified, then a *layout* configuration file whose name derive from the loaded preset filename will be created. At each later loading of this preset, the *Control Modules* arrangement will then be restored.

The **Preset Modes** menu enables to create or modify (using the *Preset manager*) *presets* that are XML files defining a set of actuators and detectors used for a given experiment. Each experiment has therefore a corresponding preset file. At startup, the program checks for existing preset files and create a menu entry for each of them.

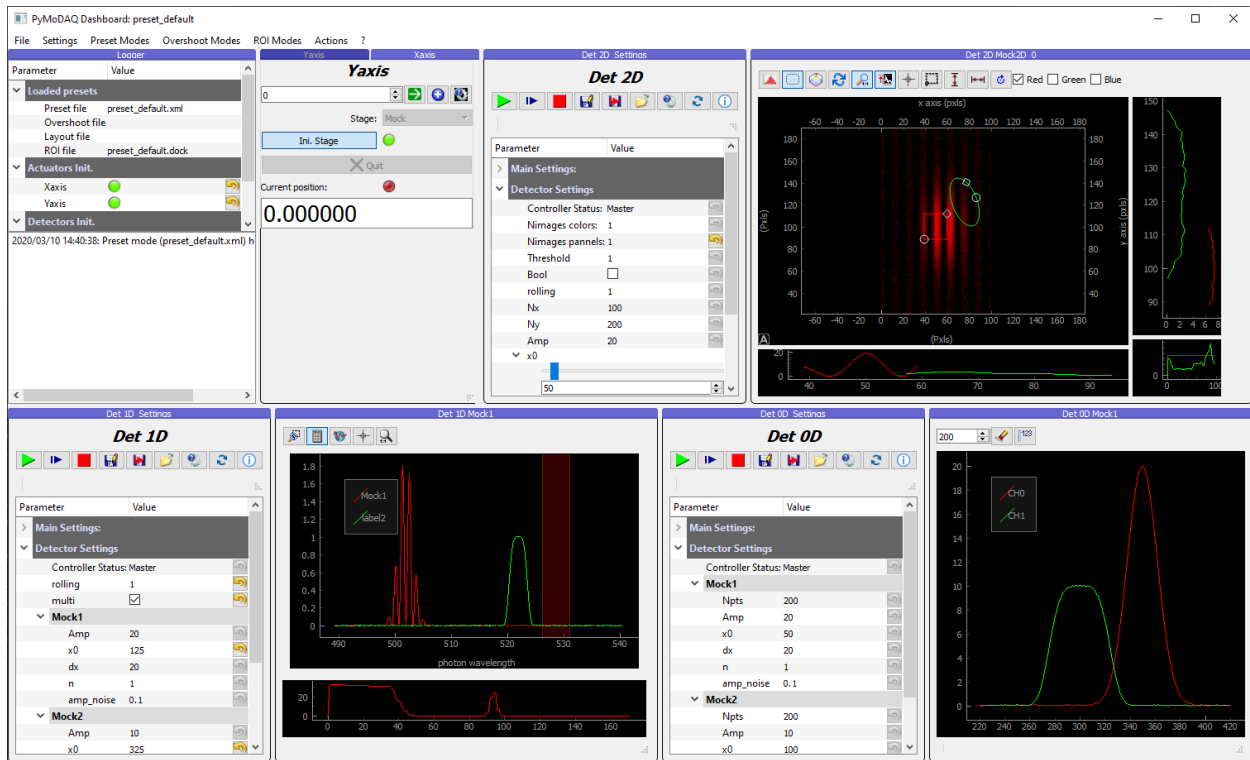


Fig. 7.10: Dashboard user interface containing all declared control modules (actuators/detectors) and some initialization info.

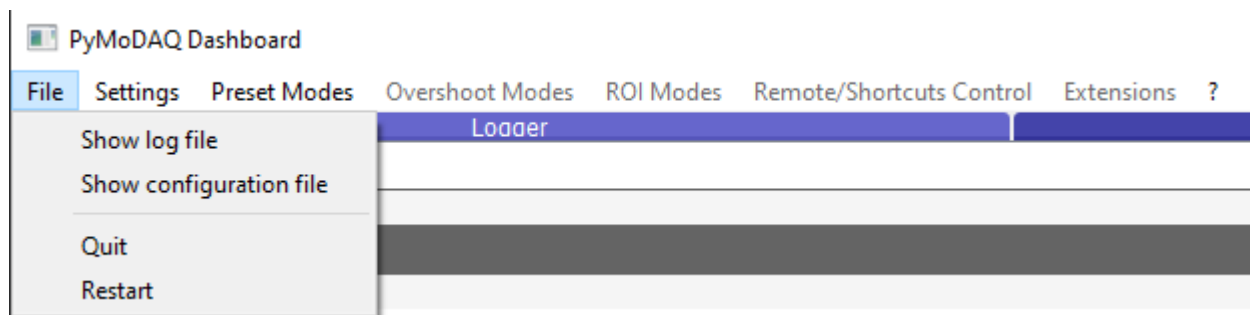


Fig. 7.11: Dashboard menu bar.

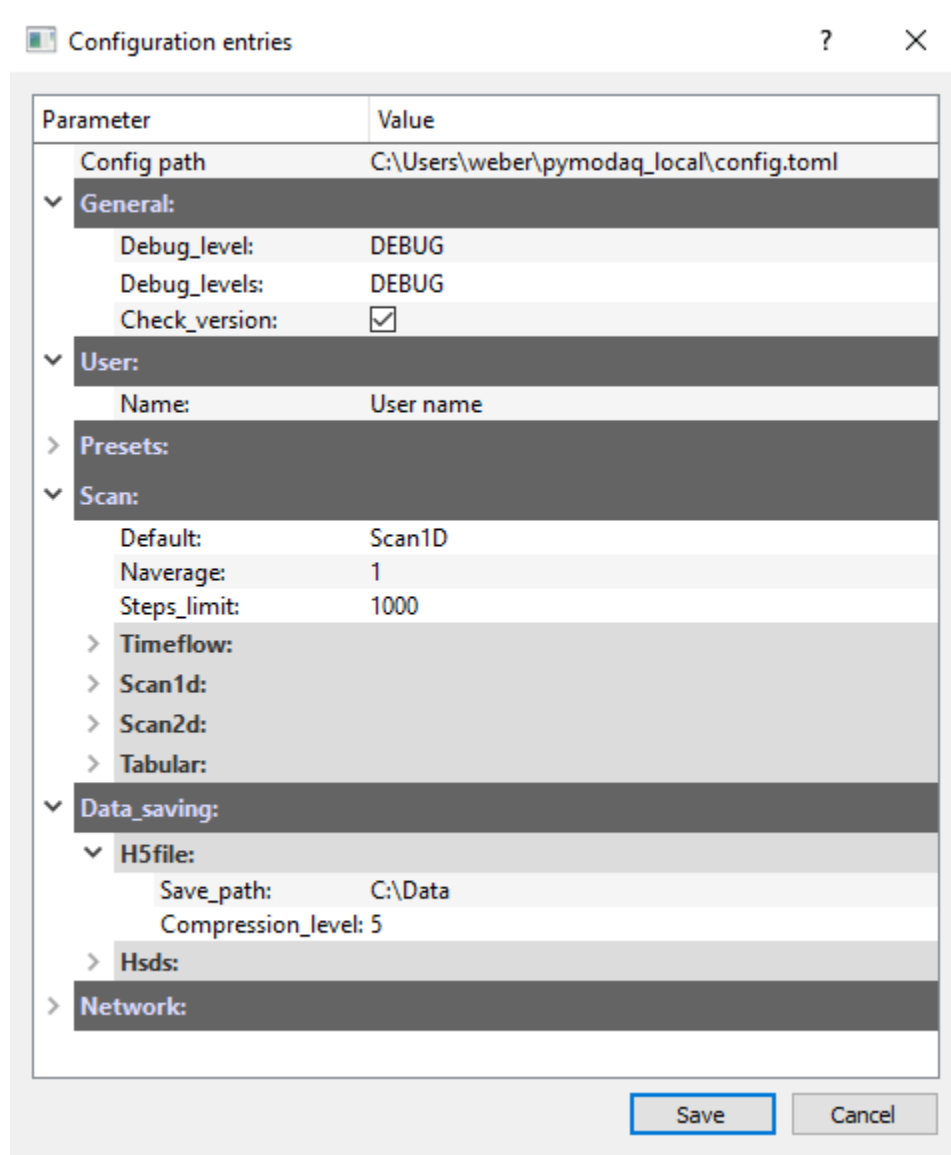


Fig. 7.12: Configuration popup window.

The **Overshoot Modes** menu is used to configure actions like stopping the acquisition or setting the value of a given actuator when a detected value (from a running detector module) gets out of range with respect to some predefined bounds. For details, see [Overshoot manager](#).

The **ROI Modes** menu, see [ROI manager](#), is used to save the state of all regions of interest defined by a user within the 1D or 2D viewers declared in the *DAQ_Viewers* control modules in the *Dashboard*. You can then, in one go, recall a particular complex configuration for data acquisition.

The **Remote/Shortcuts Control** menu, see [Remote Manager](#), is used to define key sequences on a keyboard or buttons/joysticks on a gamepad to trigger specific actions from the *Control modules*, for instance jogging of the actuator values using a joystick or grabbing data from a detector using a button.

The **Extensions** menu let the user load a specific installed extensions. Default ones are the *DAQ_Scan* and *DAQ_Logger* ones. More specific ones can be installed, for instance the package [Pymodaq Femto](#)

Multiple hardware from one controller

Sometimes one hardware controller can drive multiple actuators and sometimes detectors (for instance a XY translation stage). For this particular case the controller should not be initialized multiple times. One should identify one actuator referred to as *Master* and the other ones will be referred to as *Slave*. They will share the same controller address represented in the settings tree by the *Controller ID* entry. These settings will be activated within the plugin script where one can define a unique identifier for each actuator (U or V for the conex in [Fig. 7.16](#)). This feature can be enabled for both *DAQ_Move* and *DAQ_Viewer* modules but will be most often encountered with actuators, so see for more details: [Multiaxes controller](#). This has to be done using the Preset Manager

Control Modules

DAQ_Move and *DAQ_Viewer* can be used as stand alone user interface to manually control hardware. *DAQ_Viewer* can be used like this to monitor and/or log data from a specific detector. However, the main use is through the *DashBoard* module and its extensions (such as the [DAQ Scan](#) that will be used to perform automatic data acquisition)

DAQ Move

This module is to be used to control any [Actuator](#) hardware. An *Actuator* is, in a general sense, any parameter that one can control and may vary during an experiment. The default actuator is a Mock one (a kind of software based actuator displaying a *position* and accepting absolute or relative *positioning*).

Introduction

This module has a generic interface in the form of a dockable panel containing the interface for initialization, the manual control of the actuator *position* and a side tree like interface displaying all the settings. [Fig. 7.13](#) shows the minimal interface of the module (in order to take minimal place in the Dashboard)

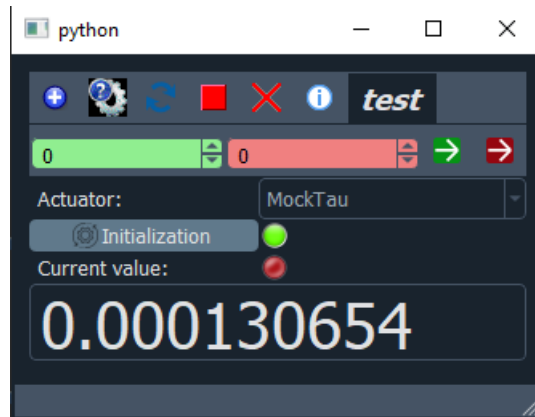
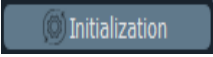



Fig. 7.13: Minimal DAQ_Move user interface

Hardware initialization

- **Actuator:** list of available instrument plugins of the DAQ_Move type, see Fig. 7.14.
-  **Initialization**: Initialize the hardware with the given settings (see *Instrument Plugins* for details on how to set hardware settings.)
- : De-initialize the hardware and quit the module

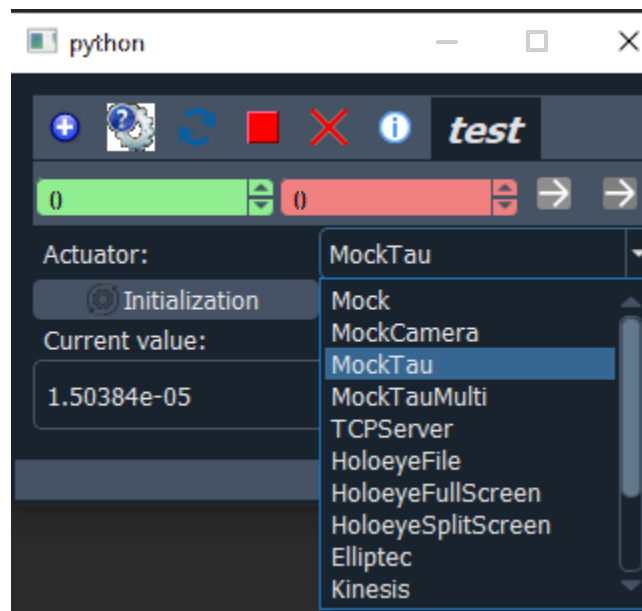





Fig. 7.14: Menu list displaying the available instrument plugin of type DAQ_Move

Positioning

Once the hardware is initialized, the actuator's *value* is displayed on the *Current value* display (bottom of Fig. 7.13) while the absolute *value* can be set using one of the top spinbox (respectively green or red) and apply it using respectively the  or  button. This double positioning allows to quickly define two values and switch between them.

Advanced positioning

More options can be displayed in order to precisely control the actuator by pressing the  button. The user interface will then look like Fig. 7.15.

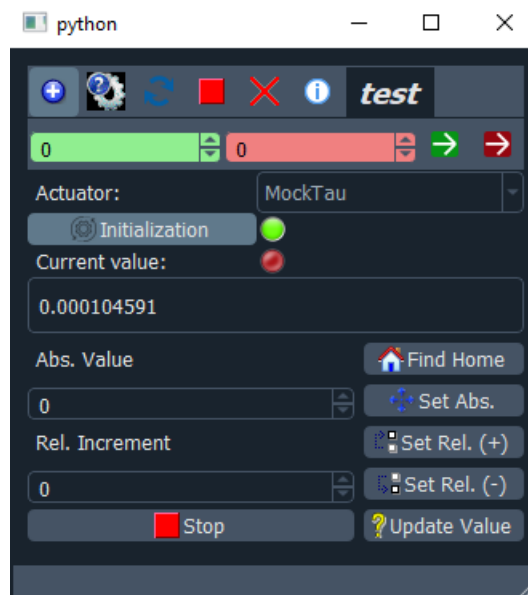
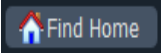
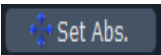
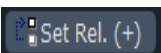
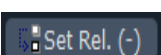
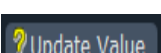
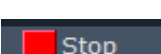



Fig. 7.15: DAQ_Move user interface with finer controls

The two new displayed spinbox relate to *Absolute* positioning and *Relative* one.

- : the actuator will try to reach a home position (known position or physical switch limit)
- : the actuator will try to reach the set *absolute* position
- : the actuator will try to reach a *relative* position (+increment)
- : the actuator will try to reach a *relative* position (-increment)
- : will update the current actuator's value display
- : stop the current motion (if possible)

Settings

The hardware and module settings can be displayed by pressing the  button. The user interface will then look like Fig. 7.16.

In the settings tree, there is two sections. The first relates to the *Main settings* of the actuator while the second relates to the hardware settings (the ones the hardware will need in order to initialize...). There is also specific settings explained below.

(not much there for the moment apart for the selected stage type and *Controller ID* that is related to multi-axes controller.

Main Settings

- *Actuator type*: is recalling the instrument plugin class being selected
- *Actuator name*: is the name as defined in the preset (otherwise it is defaulted to *test*)
- *Controller ID*: is related to multi-axes controller (see [Multiaxes controller](#))
- *Refresh value*: is the timer duration when grabbing the actuator's current value (see [Grabbing the actuator's value](#)).

Multiaxes controller

Sometimes one hardware controller can drive multiple actuators (for instance a XY translation stage). In the simplest use case, one should just initialize the instrument plugin and select (in the settings) which *axis* to use, see Fig. 7.17.

Then the selected axis can be driven normally and you can switch at any time to another one.

It is more complex when you want to drive two or more of these multi-axes during a scan. Indeed, each one should be considered in the Dashboard as one actuator. But if no particular care is taken, the Dashboard will try to initialize the controller multiple times, but only one communication channel exists, for instance a COM port. The solution in PyMoDAQ is to identify one actuator (one axis) as *Master* and the other ones will be referred to as *Slave*. They will share the same controller address (and actual driver, wrapper, ...) represented in the settings tree by the *Controller ID* entry. These settings will be activated within the instrument plugin class where one can define a unique identifier for each actuator (U or V for the conex in Fig. 7.16).

- **Controller ID**: unique identifier of the controller driving the stage
- **is Multiaxes**: boolean
- **Status**: Master or Slave
- **Axis**: identifier defined in the plugin script

These settings are really valid only when the module is used within the Dashboard framework that deals with multiple modules at the same time as configured in the [Preset manager](#) interface.

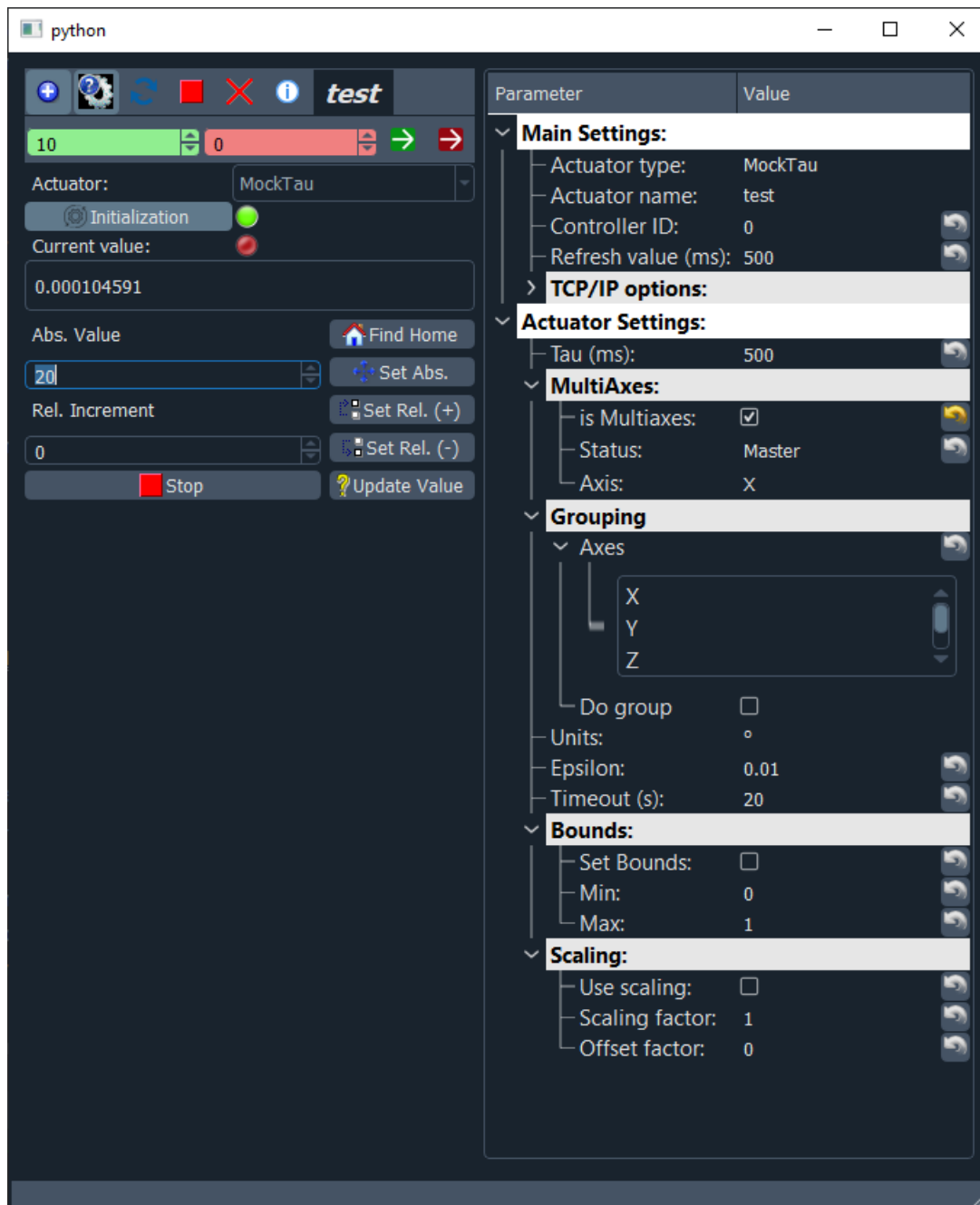


Fig. 7.16: Full DAQ_Move user interface with controls and settings

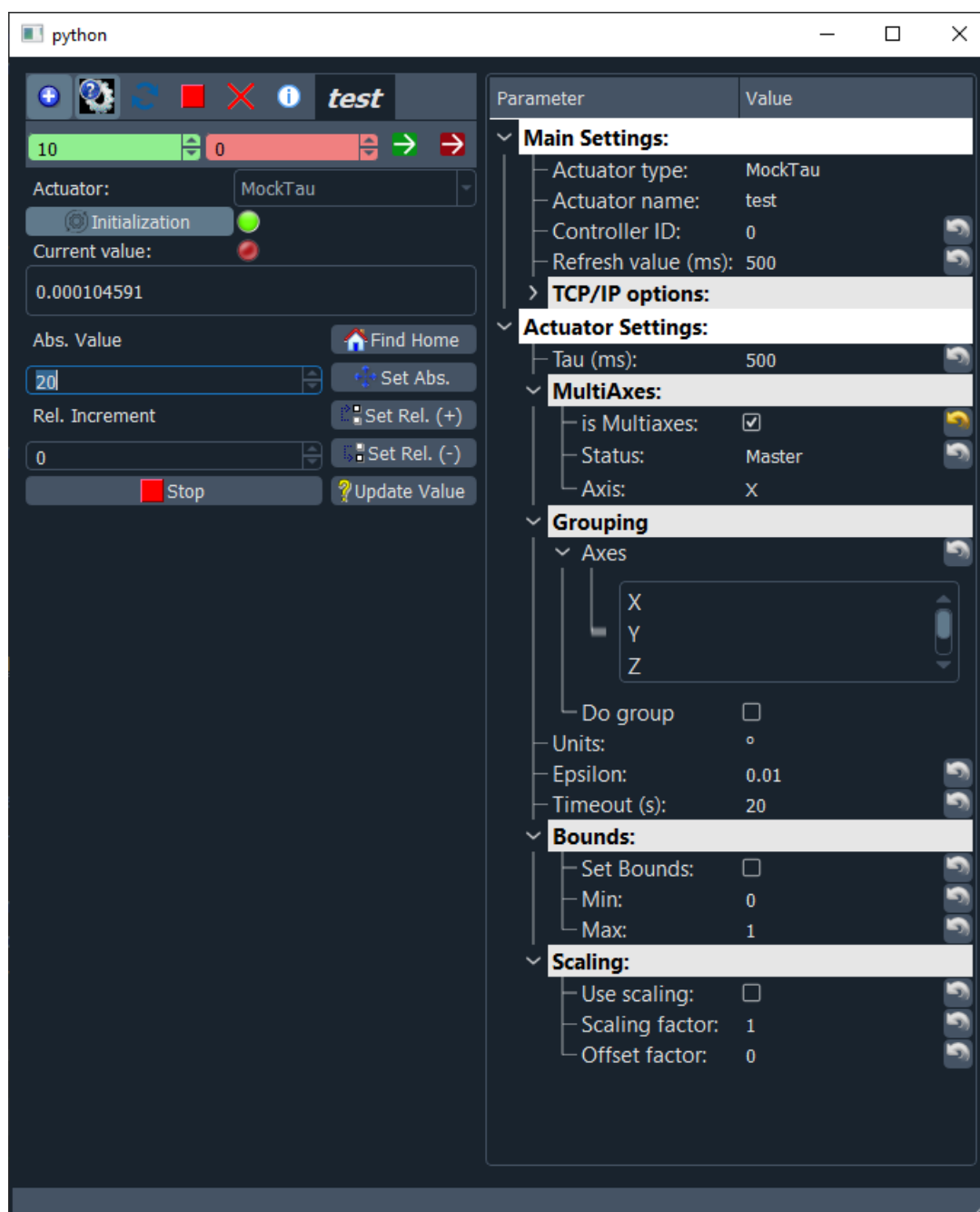


Fig. 7.17: Selection of one of the axis this controller is able to drive.

Bounds

if this section is activated (by clicking the *Set Bounds* entry) then the actuator *positions* will be software limited between *min* and *max*. This can be used to prevent the actuator to reach dangerous values for the experiment or anything else.

Scaling

If this section is activated (by clicking the *Use scaling* entry) then the *set* and *displayed* positions will be scaled as:

$$\text{new_position} = \text{scaling} * \text{old_position} + \text{offset}$$

This can be useful for instance when one deals with translation stage used to delay a laser pulse with respect to another. In that case it is easier to work with temporal units such as *femtoseconds* compared to *mm* or other native controller unit.

Other settings

- **epsilon: -very important feature-** the actuator will try to reach the target position with a precision *epsilon*. So one could use it if one want to be sure the actuator really reached a given position before moving on. However if the set precision is too small, the actuator may never reached it and will issue a timeout
- **Timeout:** maximum amount of time the module will wait for the actuator to reach the desired position.

Grabbing the actuator's value

DAQ Viewer

This module is to be used to interface any *detector*. It will display hardware settings and display data as exported by the hardware plugins (see *Emission of data*). The default detector is a Mock one (a kind of software based detector generating data and useful to test the program development). Other detectors may be loaded as plugins, see *Instrument Plugins*.

Introduction

This module has a generic interface comprised of a dockable panel related to the settings and one or more data viewer panels specific of the type of data to be acquired (see *Plotting Data*). For instance, [Fig. 7.18](#) displays a typical DAQ_Viewer GUI with a settings dockable panel (left) and a 2D viewer on the right panel.

Settings

The settings panel is comprised of 3 sections, the top one (red rectangle) displays a toolbar with buttons to grab/snap data, save them, open other settings sections and quit the application. Two types of settings can be shown/hidden: for hardware choice/initialization (green rectangle) and advanced settings to control the hardware/software (purple rectangle).

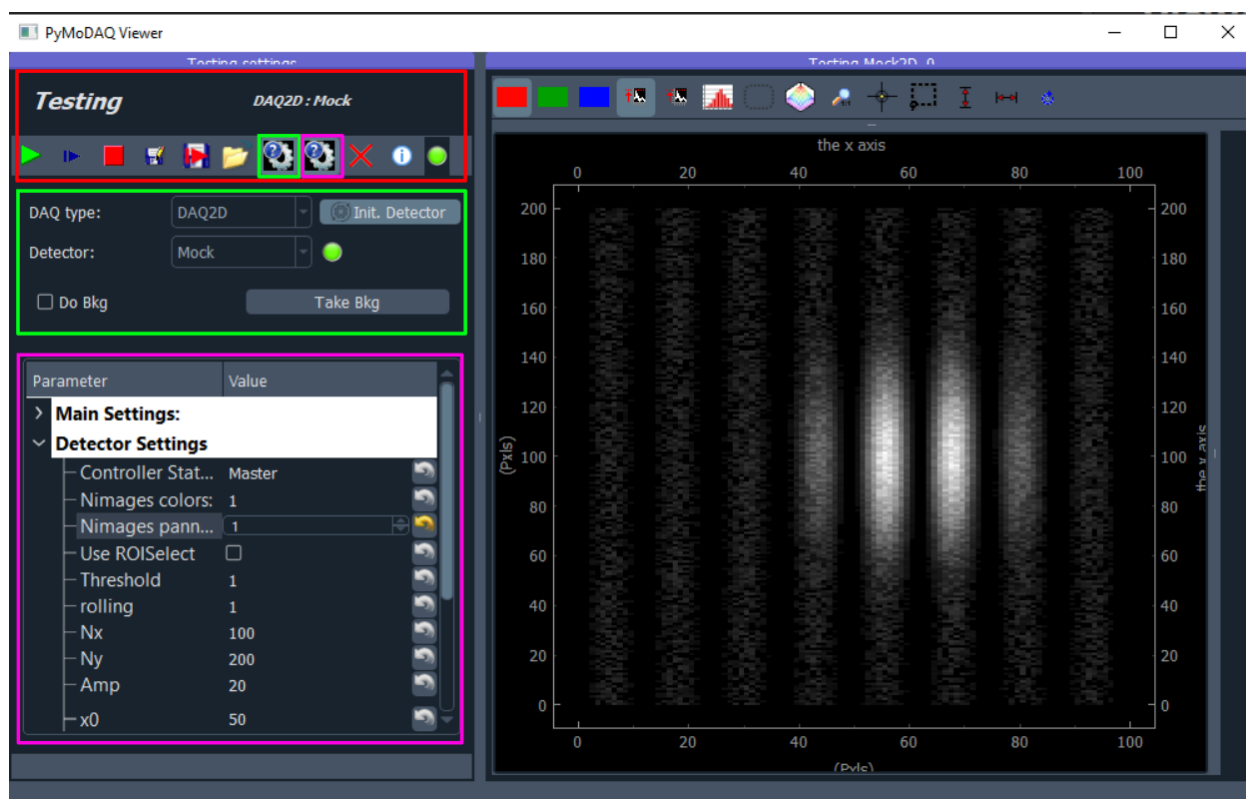


Fig. 7.18: Typical DAQ_Viewer GUI with a dockable panel for settings (left) and a 2D data viewer on the right panel. Red, green and purple rectangles highlight respectively the toolbar, the initialization and hardware settings.

Toolbar

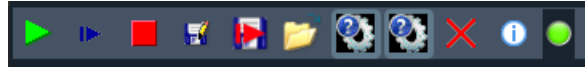












Fig. 7.19: DAQ_Viewer toolbar

The toolbar, Fig. 7.19 allows data acquisition and other actions as described below:

- : Start a continuous grab of data. Detector must be initialized.
- : Start a single grab (snap). Strongly advised for the first time data is acquired after initialization.
- : Save current data
- : Do a new snap and then save the data
- : Load data previously saved with the save button
- : Display or hide initialization and background settings
- : Display or hide hardware/software advanced settings
- : quit the application
- : open the log in a text editor
- : LED reflecting the data grabbed status (green when data has been taken)

Hardware initialization

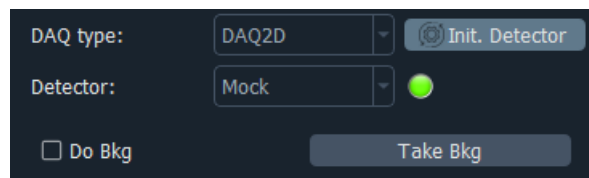
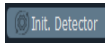
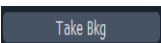
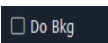


Fig. 7.20: Hardware choice, initialization and background management

The second section, Fig. 7.20 allows the choice of the instrument plugin of type detector selection. They are subdivided by dimensionality of the data they are generating (DAQ2D for cameras, DAQ1D for waveforms, timeseries... and DAQ0D for detectors generating scalars such as powermeter, voltmeter...). Once selected, the  button will start the initialization using eventual advanced settings. If the initialization is fine, the corresponding LED will turn green and you'll be able to snap data or take background:

- : do a specific snap where the data will be internally saved as a background (and saved in a hdf5 file if you save data)
- : use the background previously snapped to correct the displayed (only displayed, saved data are still raw data) data.

The last section of the settings (purple rectangle) is a ParameterTree allowing advanced control of the UI and of the hardware.

Main settings

Main settings refers to settings common to all instrument plugin. They are mostly related to the UI control.

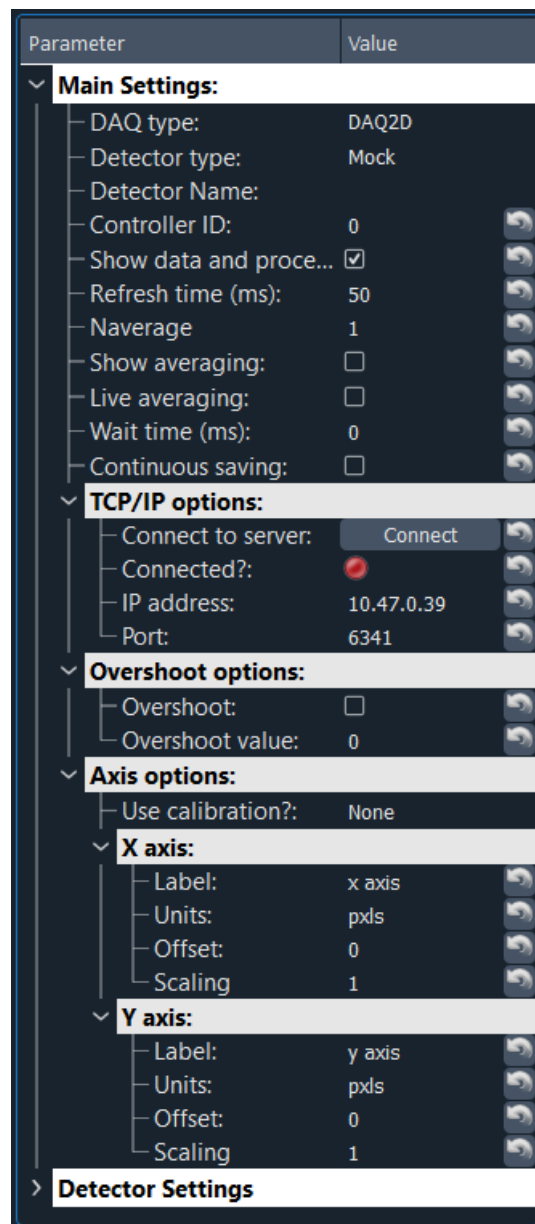


Fig. 7.21: Typical DAQ_Viewer *Main settings*.

- **DAQ type:** readonly string recalling the DAQ type used
- **Detector type:** readonly string recalling the selected plugin

- **Detector Name:** readonly string recalling the given name of the detector (from the preset)
- **Controller ID:** integer used to deal with a controller controlling multiple hardware, see [Multiple hardware from one controller](#)
- **Show data and process:** boolean for plotting (or not data in the data viewer)
- **Refresh time:** integer used to slow down the refreshing of the display (but not of the eventual saving...)
- **Naverage:** integer to set in order to do data averaging, see [Hardware averaging](#).
- **Show averaging:** in the case of software averaging (see [Hardware averaging](#)), if this is set to True, intermediate averaging data will be displayed
- **Live averaging:** *show averaging* must be set to False. If set to True, a *live grab* will perform non-stop averaging (current averaging value will be displayed just below). Could be used to check how much one should average, then set *Naverage* to this value
- **Wait time (ms):** Extra waiting time before sending data to viewer, can be used to cadence DAQ_Scan execution, or data logging
- **Continuous saving:** useful for data logging. Will display new options below in order to set a h5 file to log live data, see [Continuous Saving](#).
- **Overshoot options:** useful to protect the experiment. If this is activated, then as soon as any value of the datas exported by this detector reaches the *overshoot value*, the module will throw a *overshoot_signal* (boolean PyQtSignal). The overshoot manager of the *Dashboard* generalize this feature (see [Overshoot manager](#)) by triggering actions on actuators if overshoot signals are detected. Other features related will soon be added (action triggered on a DAQ_Move, for instance a shutter on a laser beam)
- **Axis options:** only valid for 2D detector. You can add labels, units, scaling and offset (with respect to pixels) to both x and y axis of the detector. Redundant with the plugin data export feature (see [Emission of data](#))

Data Viewers

Data Viewers presented in section [Plotting Data](#) are the one used to display data from detectors controlled from the DAQ_Viewer. By default, one viewer will be set with its type (0D, 1D, 2D, ND) depending on the detector main dimensionality (DAQ_type: DAQ0D, DAQ1D, DAQ2D...) but in fact the data viewers are set depending on the data exported from the detector plugin using the *data_grabed_signal* or *data_grabed_signal_temp* signals.

These two signals emit a list of *DataFromPlugins* objects. The **length** of this list will set the **number of dedicated data viewers**. In general one, but think about data from a Lockin amplifier generating an amplitude in volt and a phase in degrees. They are unrelated physical values better displayed in separated axes or viewers. The *DataFromPlugins*'s attribute *dim* (a string either equal to *Data0D*, *Data1D*, *Data2D*, *DataND*) will determine the data viewer type to set.

This code in a plugin

```
self.data_grabed_signal.emit([
    DataFromPlugins(name='Mock1', data=data1, dim='Data0D'),
    DataFromPlugins(name='Mock2', data=data2, dim='Data2D')])
```

will trigger two separated viewers displaying respectively 0D data and 2D data.

Other utilities

There are other functionalities that can be triggered in specific conditions. Among those, you'll find:

- The LCD screen to display 0D Data
- The ROI_select button and ROI on a Viewer2D

Saving data

Data saved from the DAQ_Viewer are data objects has described in *What is PyMoDAQ's Data?* and their saving mechanism use one of the objects defined in *Module Savers*. There are three possibilities to save data within the DAQ_Viewer.

- The first one is a direct one using the snapshots buttons to save current or new data from the detector, it uses a DetectorSaver object to do so. The private method triggering the saving is `_save_data`.
- The second one is the continuous saving mode. It uses a DetectorEnlargeableSaver object to *continuously* save data within enlargeable arrays. Methods related to this are: `append_data` and `_init_continuous_save`
- The third one is not used directly from the DAQ_Viewer but triggered by extensions such as the DAQ_Scan. Data are indexed within an already defined array using a DetectorExtendedSaver. Methods related to this are: `insert_data` and some code in the DAQ_Scan, see below.

```
for det in self.modules_manager.detectors:
    det.module_and_data_saver = module_saving.DetectorExtendedSaver(det, self.scan_shape)
self.module_and_data_saver.h5saver = self.h5saver # will update its h5saver and all
↳ submodules's h5saver
```

Snapshots

Datas saved directly from a DAQ_Viewer (for instance the one on [Fig. 7.27](#)) will be recorded in a h5file whose structure will be represented like [Fig. 7.71](#) using PyMoDAQ's h5 browser.

Continuous Saving

When the *continuous saving* parameter is set, new parameters are appearing on the *DAQ_Viewer* panel (see [Fig. 7.22](#)). This is in fact the settings associated with the H5Saver object used under the hood, see *H5Saver*.

- *Base path*: indicates where the data will be saved. If it doesn't exist the module will try to create it
- *Base name*: indicates the base name from which the save file will derive
- *Current Path*: *readonly*, complete path of the saved file
- *Do Save*: Initialize the file and logging can start. A new file is created if clicked again.
- *Compression options*: data can be compressed before saving, using one of the proposed library and the given value of compression [0-9], see *pytables* documentation.

The saved file will follow this general structure:

```
D:\Data\2018\20181220\Data_20181220_16_58_48.h5
```

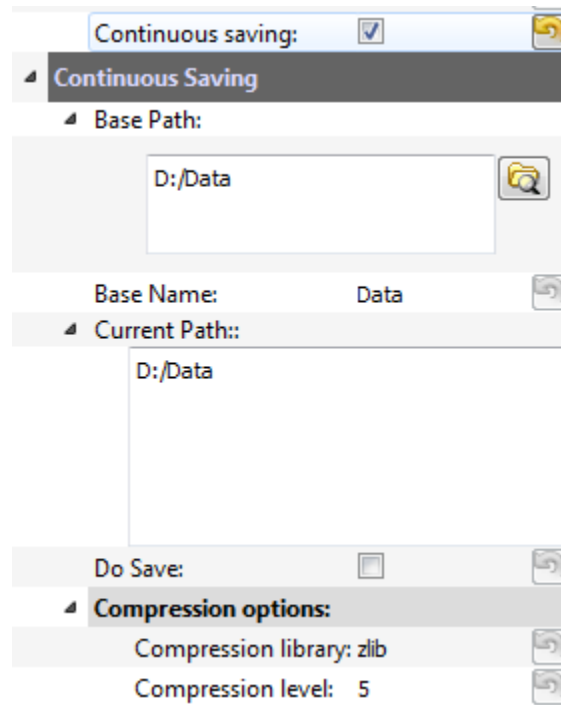


Fig. 7.22: Continuous Saving options

With a base path (D:\Data in this case) followed by a subfolder year, a subfolder day and a filename formed from a *base name* followed by the date of the day and the time at which you started to log data. Fig. 7.23 displays the tree structure of such a file, with two nodes (prefixed as enlargeable, *EnlData*) and a navigation axis corresponding to the timestamps at the time of each snapshot taken once the continuous saving has been activated (ticking the Do Save checkbox)

7.3.5 Extensions

The DashBoard module can load extensions to perform dedicated tasks, such as automated data acquisition.

DAQ Scan

This module is an extension of the DashBoard but is the heart of PyMoDAQ, it will:

- setup automatic data acquisition of detectors as a function of one or more actuators
- save datas in hierarchical hdf5 binary files (compatible with the [H5Browser](#) used to display/explore data)

The flow of this module is as follow:

- at startup you have to define/load a preset (see [Preset manager](#)) in the Dashboard
- Select DAQ_Scan in the actions menu
- A dataset will be declared the first time you set a scan. A dataset is equivalent to a single saved file containing multiple scans. One can see a dataset as a series of scans related to single *subject/sample to be characterized*.
- Metadata can be saved for each dataset and then for each scan and be later retrieved from the saved file (see [Module Savers](#) and [H5Browser](#))

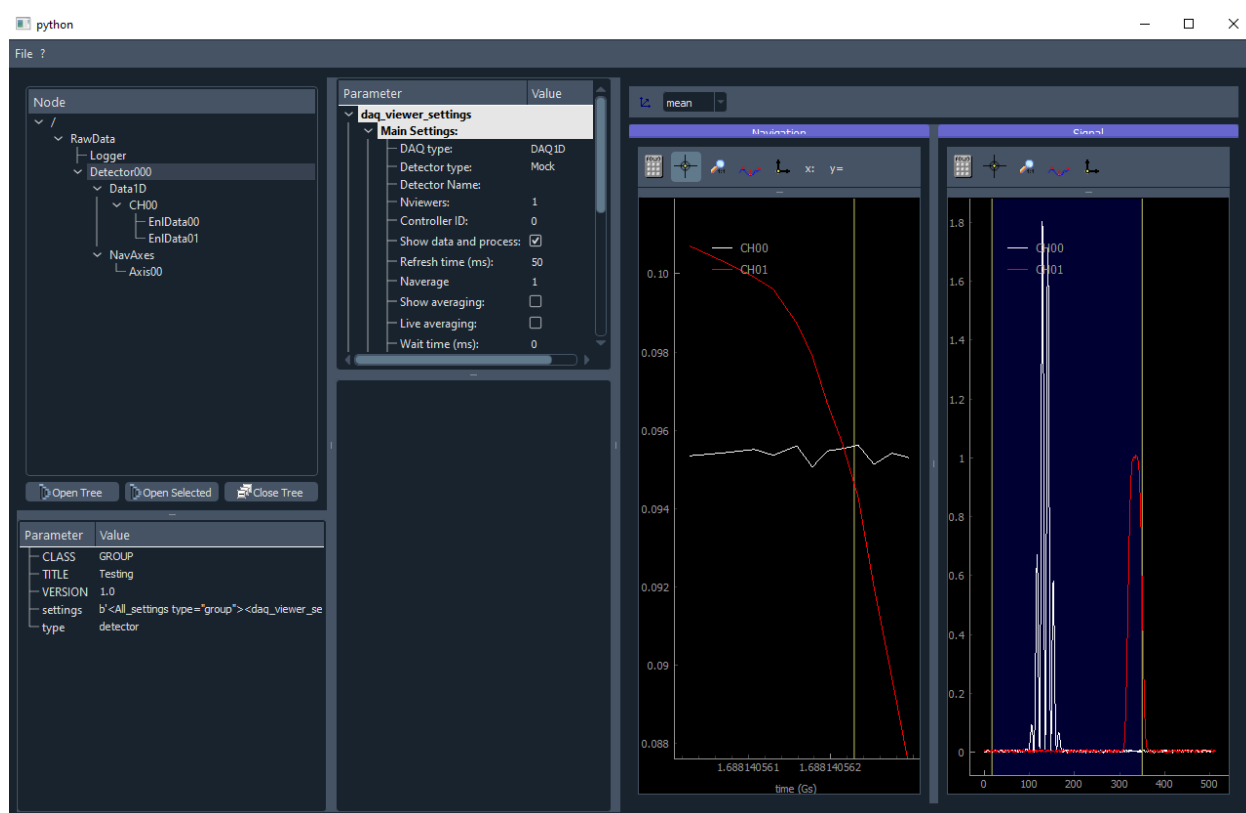


Fig. 7.23: Continuous Saving options

- Performs multiple scans exploring all the parameters needed for your experiment

Introduction

The dashboard gives you full control for manual adjustments (using the UI) of each actuator, checking their impact on live data from the detectors. Once all is set, one can move to an automated scan using the main control window of the DAQ_Scan, see Fig. 7.24.

Main Control Window

The main control window is comprised of various panels to set all parameters and display live data taken during a scan.

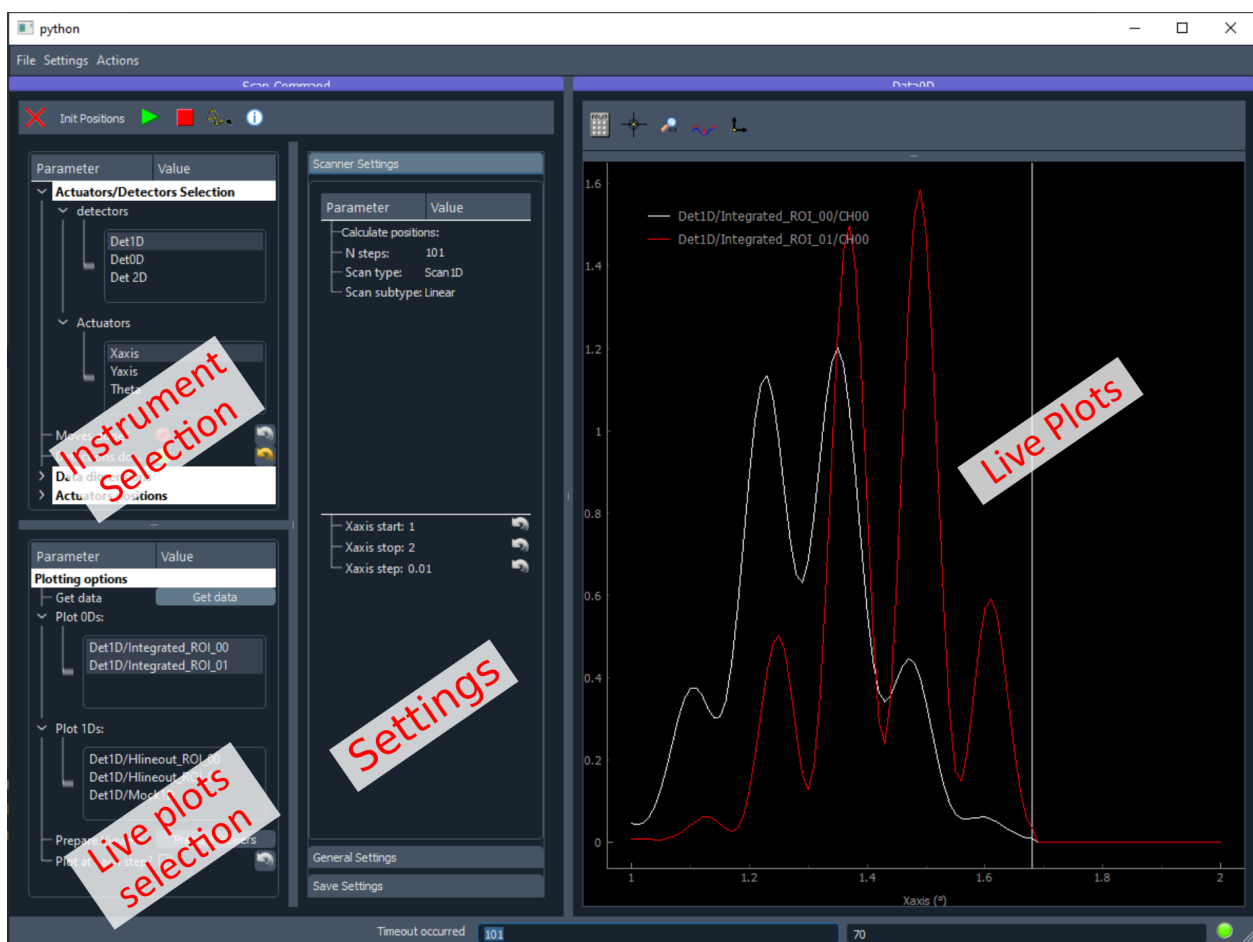


Fig. 7.24: Main DAQ_Scan user interface.

- The *instrument selection* panel allows to quickly select the detectors and the actuators to use for the next scan
- The *settings* panel is divided in three sections (see [Settings](#) for more details):
 - Scanner settings: select and set the next scan type and values.
 - General settings: options on timing, scan averaging and plotting.

- Save settings: everything about what should be saved, how and where.
- The *Live plots selection* panel allows to select which data produced from selected detectors should be rendered live
- The *Live Plots* panels renders the data as a function of varying parameters as selected in the *Live plots selection* panel

Scan Flow

Performing a scan is typically done by:

- Selecting which detectors to save data from
- Selecting which actuators will be the scan varying parameters
- Selecting the type of scan (see *Selecting the type of scan*): 1D, 2D, ... and subtypes
- For a given type and subtype, settings the start, stop, ... of the selected actuators
- Selecting data to be rendered live (none by default)
- Starting the scan

Selecting detectors and actuators

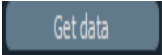
The *Instrument selection* panel is the user interface of the module manager (see *Module Manager* for details). It allows the user to select the actuators and detectors for the next scan (see Fig. 7.25). This interface is also used for the DAQ_Logger extension.

Selecting the type of scan

All specifics of the upcoming scan are configured using the scanner_paragraph module interface as seen on Fig. 7.26 in the case of a spiral Scan2D scan configuration.

Selecting the data to render live

For a data acquisition system to be efficient, live data must be plotted in order to follow the experiment behaviour and check if something is going wrong or successfully without the need to perform a full data analysis. For this PyMoDAQ live data display will allows the user to select data to be plotted from the selected detectors.

The list of all possible data to be plotted can be obtained by clicking on the  button. All data will be classified by dimensionality (0D, 1D). The total dimensionality of the data + the scan dimensions (1 for scan1D and 2 for Scan2D...) should not exceed 2 (this means one cannot plot more complex plots than 2D intensity plots). It also means that you should use ROI to generate lower dimensionality data from your raw data for a proper live plot.

For instance, if the chosen detector is a 1D one, see Fig. 7.27. Such a detector can generate various type of live data.

It will export the raw 1D data and the 1D lineouts and integrated 0D data from the declared ROI as shown on Fig. 7.28

Parameter	Value
▼ Actuators/Detectors Selection	
▼ detectors	
	<div>Det 1D Det 0D Det 2D</div>
▼ Actuators	
	<div>Theta Axis Yaxis Xaxis</div>
Moves done?	<div><div></div><div></div></div>
Detections done?	<div><div></div><div></div></div>
> Data dimensions	
> Actuators positions	

Fig. 7.25: List of declared modules from a preset

Parameter	Value
Scanner settings	
<button>calculate_positions</button>	
N steps:	9
Scan type:	Scan2D
Scan2D settings	
Scan subtype:	Spiral
Selection:	Manual
Center Ax1	5
Center Ax2	5
Step Ax1:	1
Step Ax2:	5
Npts/axis	2
Rmax Ax1	1
Rmax Ax2	5
<button>load_xml</button>	
<button>save_xml</button>	

Fig. 7.26: The Scanner user interface set on a *Scan2D* scan type and an *adaptive* scan subtype and its particular settings.



Fig. 7.27: An example of a 1D detector having 2 channels. 0D data are generated as well from the integration of channel CH0 within the regions of interest (ROI_00 and ROI_01).

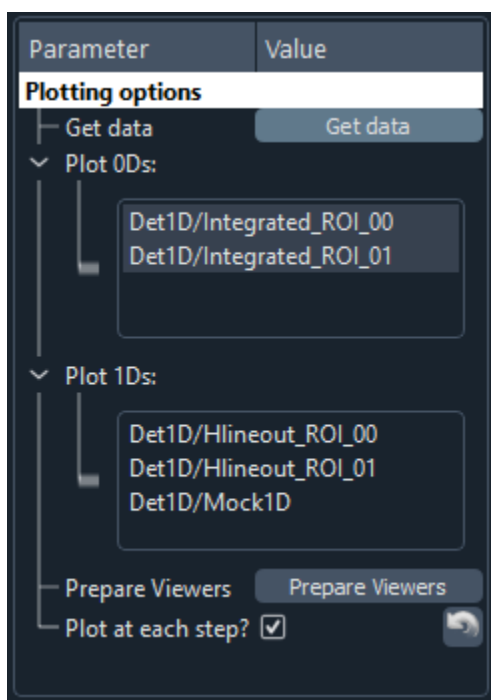


Fig. 7.28: An example of all data generated from a 1D detector having 2 channels. 0D data and 1D data are generated as well from the integration of channel CH0 within the regions of interest (ROI_00 and ROI_01).

Given these constraints, one live plot panel will be created by selected data to be rendered with some specificities. One of these is that by default, all 0D data will be grouped on a single viewer panel, as shown on [Fig. 7.24](#) (this can be changed using the *General Settings*)

The viewer type will be chosen (Viewer1D or 2D) given the dimensionality of the data to be plotted and the number of selected actuators.

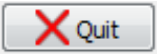




- if the scan is 1D:
 - exported 0D datas will be displayed on a Viewer1D panel as a line as a function of the actuator *position*, see [Fig. 7.24](#).
 - exported 1D datas will be displayed on a Viewer2D panel as color levels as a function of the actuator *position*, see [Fig. 7.29](#).
- if the scan is 2D:
 - exported 0D datas will be displayed on a Viewer2D panel as a pixel map where each pixel coordinates represents a scan coordinate. The color and intensity of the pixels refer to channels and data values, see [Fig. 7.30](#) for a *linear* 2D scan.

So at maximum, 2D dimensionality can be represented. In order to see live data from 2D detectors, one should therefore export lineouts from ROIs or integrate data. All these operations are extremely simple to perform using the ROI features of the data viewers (see [Plotting Data](#))

Various settings

Toolbar

The toolbar is comprised of buttons to start and stop a scan as well as quit the application. Some other functionalities can also be triggered with other buttons as described below:

- : will shut down all modules and quit the application (redundant with: *File/Quit* menu)
- **Init. Positions:** will move all selected actuators to their initial positions as defined by the currently set scan.
- : will start the currently set scan (first it will set it then start it)
- : stop the currently running scan (in case of a batch of scans, it will skip the current one).
- : when checked, allows currently actuators to be moved by double clicking on a position in the live plots
- : opens the logs in a text editor

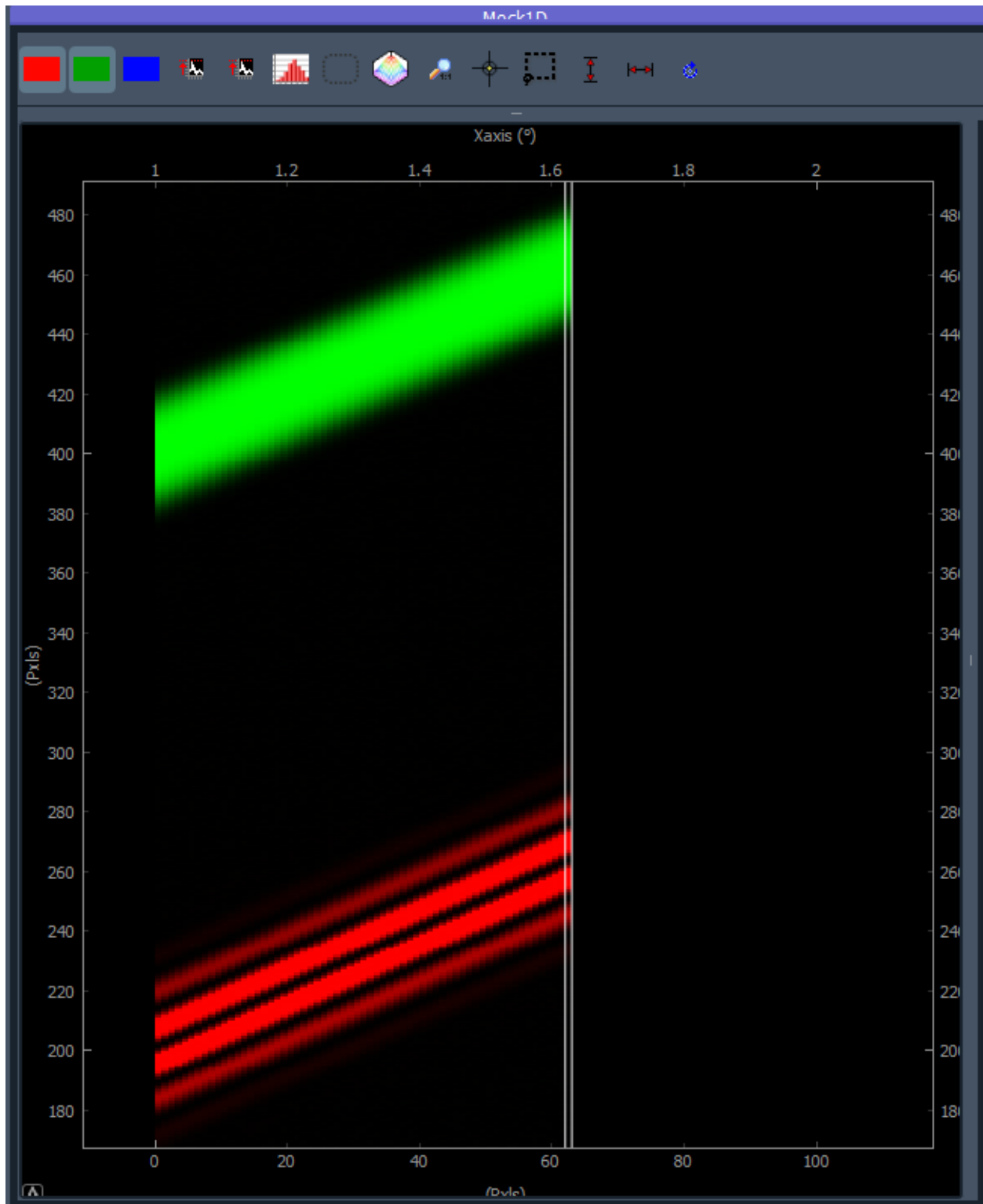


Fig. 7.29: An example of a detector exporting 1D live data plotted as a function of the actuator *position*. Channel CH0 is plotted in red while channel CH1 is plotted in green.

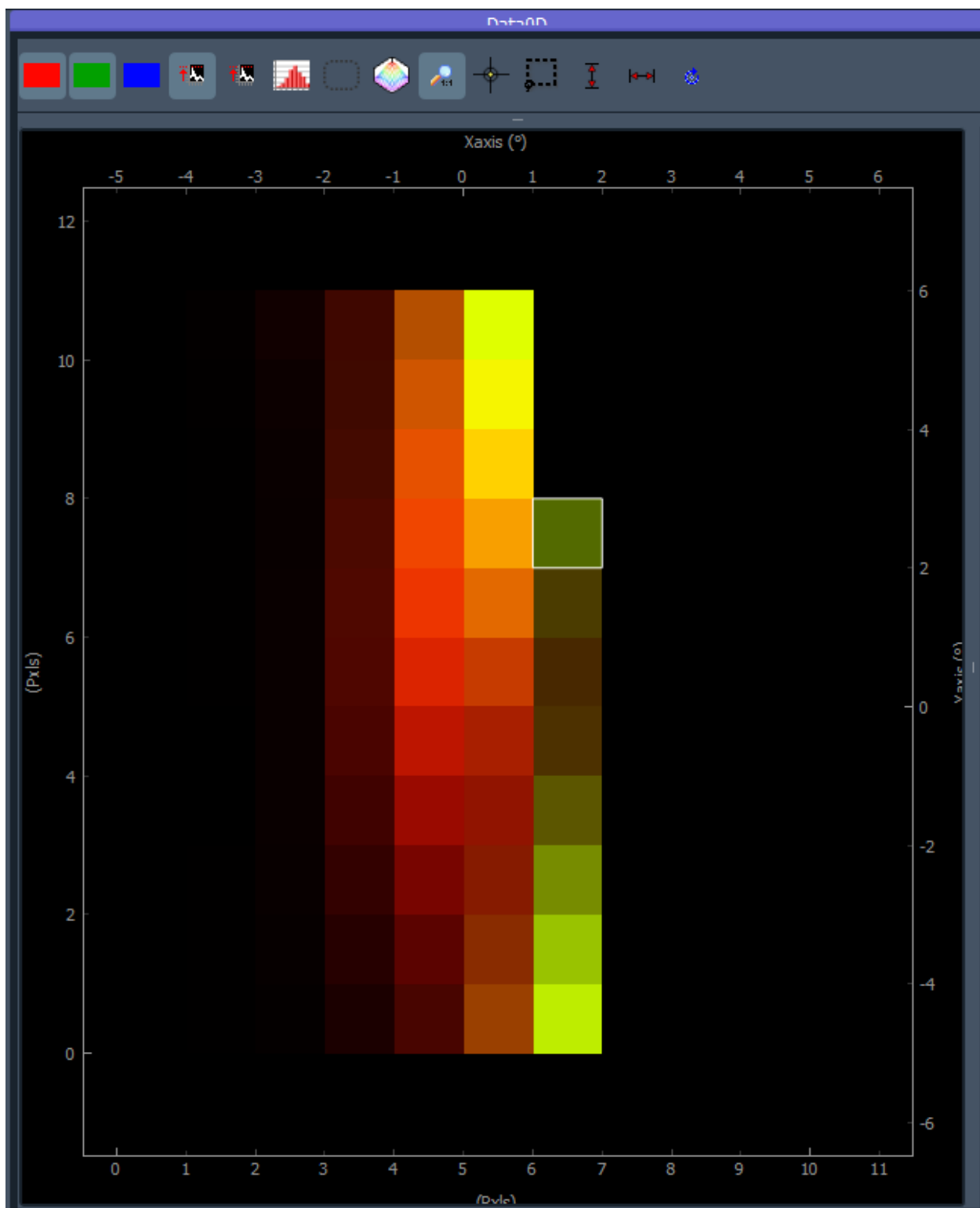


Fig. 7.30: An example of a detector exporting 0D live data plotted as a function of the 2 actuators's *position*. Integrated regions of channel CH0 are plotted in red and green.

Menu Bar Description

There are two entries in the menu bar: *File* and *Settings*

The *File* entry will let you:

- load a previously saved scan file (and keep saving scans on it)
- Save the current file in another filename than the default one
- Load the content of the current file into the *H5Browser*

The *Settings* entry will let you:

- display the *Navigator* see [Navigator](#)
- Display and activate the *Scan Batch Manager*

Settings

The settings tree as shown on [Fig. 7.24](#) is actually divided in a few subtrees that contain everything needed to define a given scan, save data and plot live information.

General Settings

The General Settings are comprised of:

- **Time Flow**
 - **Wait time step**: extra time the application wait before moving on to the next scan step. Enable rough timing if needed
 - **Wait time between**: extra time the application wait before starting a detector's grab after the actuators reached their final value.
 - **timeout**: raise a timeout if one of the scan step (moving or detecting) is taking a longer time than timeout to respond
- **Scan options :**
 - **N average**: Select how many scans to average. Save all individual scans.
- **Scan options :** * **Get Data** probe selected detectors to get info on the data they are generating (including processed data from ROI) * **Group 0D data**: Will group all generated 0D data to be plotted on the same viewer panel (work only for 0D data) * **Plot 0D** shows the list of data that are 0D * **Plot 1D** shows the list of data that are 1D * **Prepare Viewers** generates viewer panels depending on the selected data to be live plotted * **Plot at each step**
 - if checked, update the live plots at each step in the scan
 - if not, display a **Refresh plots** integer parameter, say T. Will update the live plots every T milliseconds
- **Save Settings**: See h5saver_settings

Saving: Dataset and scans

DAQ_Scan module will save your data in **datasets**. Each **dataset** is a unique h5 file and may contain multiple scans. The idea behind this is to have a unique file for a set of related data (the **dataset**) together with all the meta information: logger data, module parameters (settings, ROI...) even *png* screenshots of the various panels.

Fig. 7.31 displays the content of a typical **dataset** file containing various scans and how each data and metadata is used by the *H5Browser* to display the info to the user.

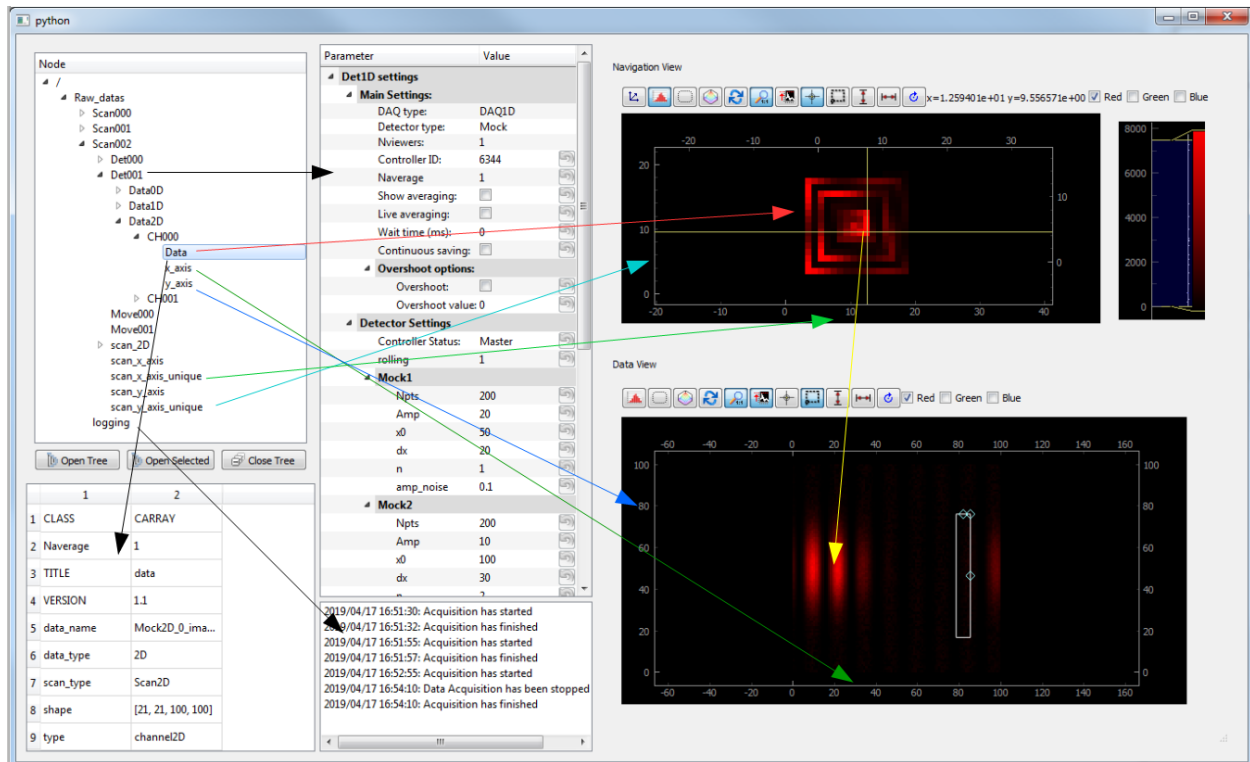


Fig. 7.31: h5 browser and arrows to explain how each data or metadata is being displayed

The Save Settings (see Fig. 7.32) is the user interface of the *H5Saver*, it is a general interface to parametrize data saving in the hdf5 file:

In order to save correctly your datas, saving modules are to be used, see *Module Savers*.

Scanner

The *Scanner* module is an object dealing with configuration of scan modes and is mainly used by the DAQ_Scan extension. It features a graphical interface, see Fig. 7.34, allowing the configuration of the scan type and all its particular settings. The **Scan type** sets the type of scan, **Scan1D** for a scan as a function of only one actuator, **Scan2D** for a scan as a function of two actuators, **Sequential** for scans as a function of 1, 2...N actuators and **Tabular** for a list of points coordinates in any number of actuator phase space. All specific features of these scan types are described below:

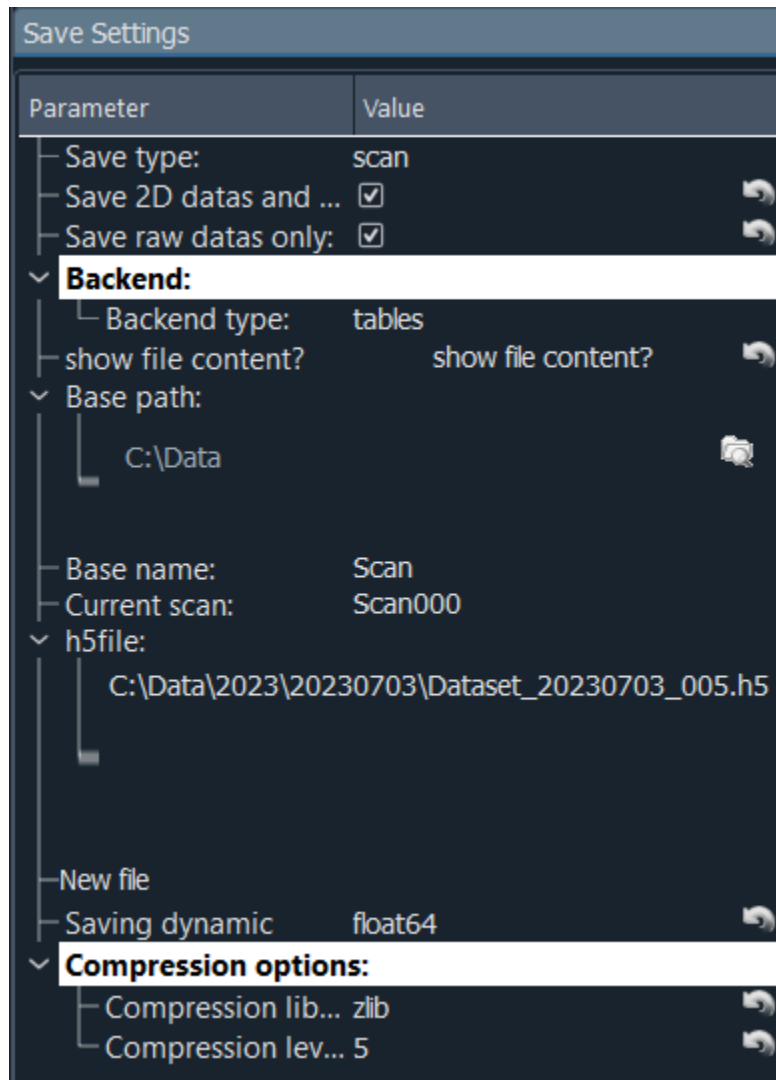


Fig. 7.32: Save settings for the DAQ_Scan extension

Scan1D

The possible settings are visible on Fig. 7.33 and described below:

- **scan subtype:** either *Linear* (usual uniform 1D scan), *Back to start* (the actuator comes back to the initial position after each linear step, for a referenced measurement for instance), *Random* same as *Linear* except the predetermined positions are sampled randomly and from version 2.0.1 *Adaptive* that features no predetermined positions. These will be determined by an algorithm influenced by the signal returned from a detector on the previously sampled positions (see *Adaptive*)
- **Start:** Initial position of the selected actuator (in selected actuator controller unit)
- **Stop:** Last position of the scan (in selected actuator controller unit)
- **Step:** Step size of the step (in selected actuator controller unit)

For the special case of the Adaptive mode, one more feature is available: the *Loss type**. It modifies the algorithm behaviour (see *Adaptive*)

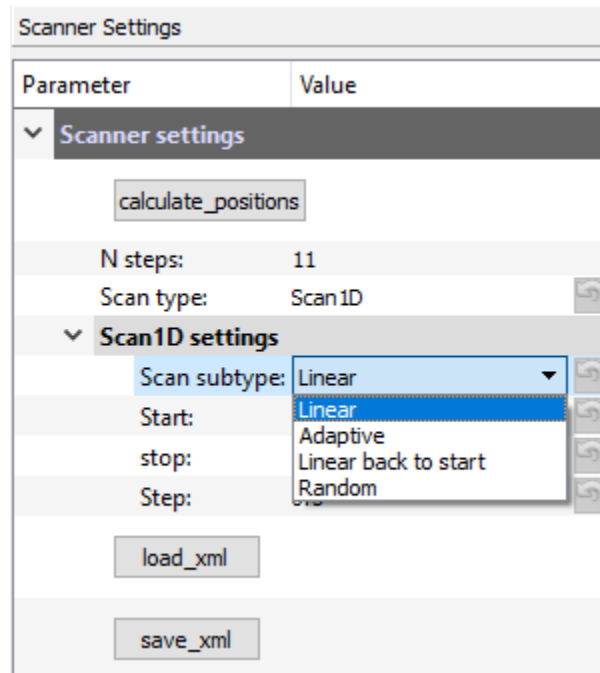


Fig. 7.33: The Scanner user interface set on a *Scan1D* scan type and the visible list of scan subtype.

Scan2D

The possible settings are visible on Fig. 7.34 and described below:

- **Scan subtype:** See Fig. 7.35 either *linear* (scan line by line), *linear back and forth* (scan line by line but in reverse direction each 2 lines), *spiral* (start from the center and scan as a spiral), *Random* (random sampling of the *linear* case) and *Adaptive* (see *Adaptive*)
- **Start, Stop, Step:** for each axes (each actuators)

Parameter	Value
Scanner settings	
<button>calculate_positions</button>	
N steps:	9
Scan type:	Scan2D
Scan2D settings	
Scan subtype:	Spiral
Selection:	Manual
Center Ax1	5
Center Ax2	5
Step Ax1:	1
Step Ax2:	5
Npts/axis	2
Rmax Ax1	1
Rmax Ax2	5
<button>load_xml</button>	
<button>save_xml</button>	

Fig. 7.34: The Scanner user interface set on a *Scan2D* scan type and a *Spiral* scan subtype and its particular settings.

- **Rmax, Rstep, Npts/axis:** in case of spiral scan only. Rmax is the maximum radius of the spiral (calculated), and Npts/axis is the number of points for both axis (total number of points is therefore $Npts/axis^2$).
- **Selection:** see [Scan Selector](#)

Scan2D subtypes

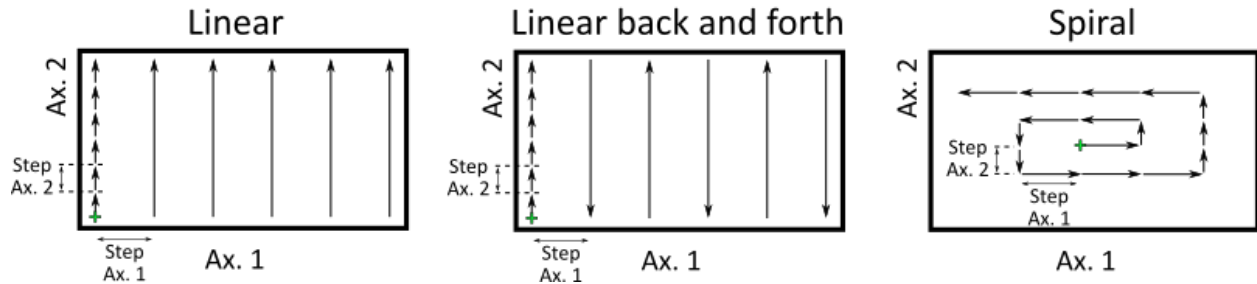


Fig. 7.35: The main Scan2D subtypes: Linear, Back and Forth and Spiral.

Sequential

The possible settings are visible on [Fig. 7.36](#) and described below:

- **Scan subtype:** only *linear* this means the scan have a sequence of Scan1D of the last specified actuator (on [Fig. 7.36](#), it is *Xaxis*) for all positions of the last but end actuator (here *Yaxis*) and so on. So on [Fig. 7.36](#) there will be 11 steps for *Xaxis* times 11 steps for *Yaxis* times 10 steps for *Theta axis* so in total $11 \times 11 \times 10 = 1210$ total steps for this 3 dimensions scan.

Note: If only 1 actuator is selected, then the Sequential scan is identical to the *Scan1D* scan but where only the *linear* subtype is available. If 2 actuators are selected, then the Sequential scan is identical to the *Scan2D* scan but where only the *linear* subtype is available.

Tabular

The tabular scan type consists of a list of positions (for each selected actuators).

Tabular Linear/Manual case

In the Linear/Manual case, the module will move actuators on each positions and grab datas. On [Fig. 7.37](#), a list of 79 positions has been set. By right clicking on the table, a context manager pops up and gives the possibility to:

- add one more position in the list
- remove the selected position
- clear all the positions
- load positions from a text file (as many columns as selected actuators with their positions separated by a tab)
- save the current list of positions in a text file (for later quick loading of positions)

One can also drag and drop elements of the list at a different index in the list.

Scanner Settings

Parameter	Value																				
▼ Scanner settings																					
<button>calculate_positions</button>																					
N steps:	1210																				
Scan type:	Sequential ▼																				
▼ Sequential settings																					
Scan subtype: Linear																					
▼ Sequences																					
	<table border="1"><thead><tr><th></th><th>Actuator</th><th>Start</th><th>Stop</th><th>Step</th></tr></thead><tbody><tr><td>0</td><td>Theta Axis</td><td>0</td><td>90</td><td>10</td></tr><tr><td>1</td><td>Yaxis</td><td>0</td><td>10</td><td>1</td></tr><tr><td>2</td><td>Xaxis</td><td>0</td><td>100</td><td>10</td></tr></tbody></table>		Actuator	Start	Stop	Step	0	Theta Axis	0	90	10	1	Yaxis	0	10	1	2	Xaxis	0	100	10
	Actuator	Start	Stop	Step																	
0	Theta Axis	0	90	10																	
1	Yaxis	0	10	1																	
2	Xaxis	0	100	10																	
< >																					
<button>load_xml</button>																					
<button>save_xml</button>																					

Fig. 7.36: The Scanner user interface set on a *Sequential* scan type with a sequence of three actuators

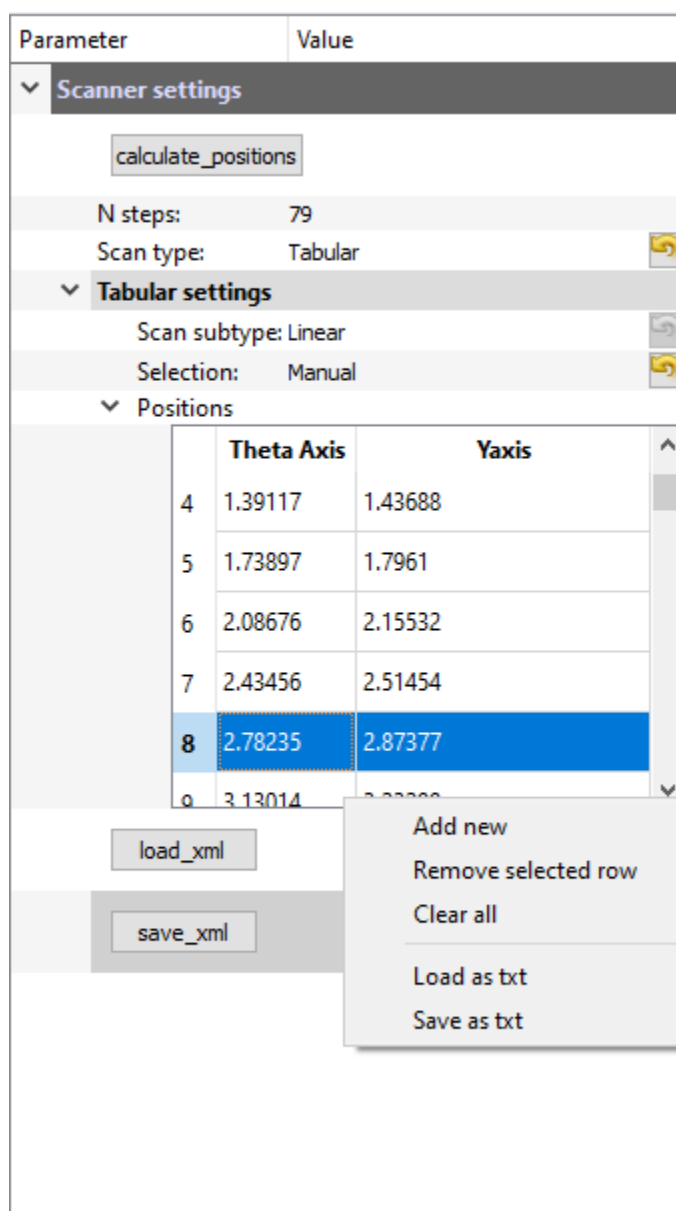


Fig. 7.37: The Scanner user interface set on a *Tabular* scan type with a list of points for 2 actuators. A context menu with other options is also visible (right click on the table to show it)

Tabular Linear/Polylines case

In the particular case of 2 selected actuators, it could be more interesting to draw the positions for the tabular scan. One possibility is to draw segments on a 2D viewer (see Fig. 7.38) and positions will be points along these segments (it will be a kind of 1D cuts within a 2D phase space). A new setting, *Curvilinear step* appears. The positions will be points starting from the start of the first segment and then step along them by the value of this setting. That gives, for Fig. 7.38, 40 points defined along the segments.

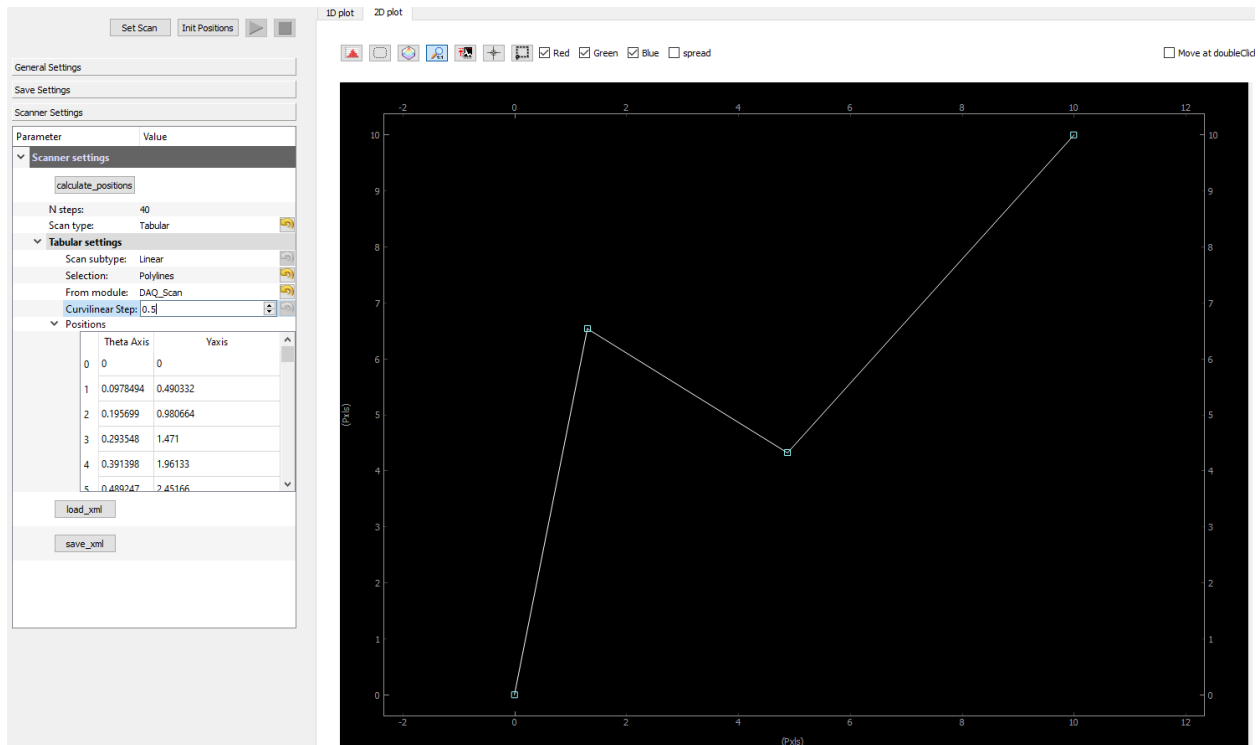


Fig. 7.38: An example of 1D complex sections selected within a 2D area

Tabular Adaptive case

Valid for 1 or 2 selected actuators. The tabular adaptive case will be similar to scan1D adaptive mode, except that one adaptive Scan1D will be done for each segments defined by the list of positions in the table. For instance, Fig. 7.39 shows a list of 4 positions defining 4 segments in a 2D space. The adaptive scan will be done on/along these 4 segments. Positions can be set manually or from a *Polylines* selection as seen on Fig. 7.38.

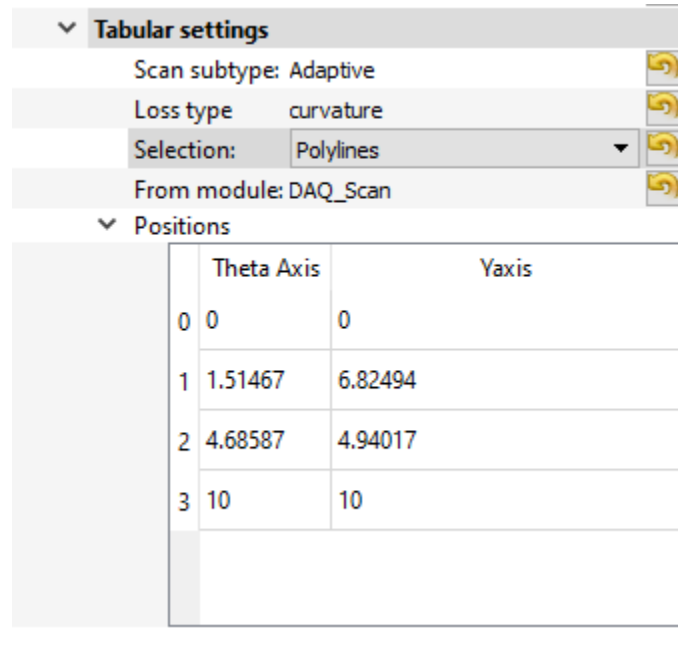


Fig. 7.39: The Scanner user interface set on a *Tabular* scan type with a list of points for 2 actuators. A context menu with other options is also visible (right click on the table to show it)

Adaptive

All the adaptive features are using the [python-adaptive](https://zenodo.org/record/1182437) package (Parallel active learning of mathematical functions, 10.5281/zenodo.1182437). And the reader is invited to explore their tutorials to discover how these algorithms work. In PyMoDAQ the [learner1D](#) algorithm is used for the *Scan1D* and *Tabular* scan types while the [learner2D](#) one is used for *Scan2D* scan type.

Bounds

As a general rule, the adaptive algorithm will need bounds to work with. For *Scan1D* scan type, these will be defined from the *start* and *stop* settings. For *Tabular*, it is the start and ends of the segments. Finally for *Scan2D*, it is the: *Start Ax 1*, *Stop Ax 1* and *Start Ax 2*, *Stop Ax 2* that are defining scan bounds.

Feedback

The adaptive algorithm will need for each probed positions a feedback value telling it the fitness of the probed points. From these on all previous points, it will determine the best next points to probe. In order to provide such a feedback, one has to choose a signal among all available from the Dashboard detectors. It has to be a Scalar so originate from a 0D detector or integrated ROI from 1D or 2D detectors. The module manager user interface (right most setting tree in the DAQ_Scan module ,see [Fig. 7.88](#)) will let you probe available datas exported from currently selected detectors. You can then pick the Data0D one you want to use as the Adaptive feedback. For instance, on [Fig. 7.88](#), three Data0D are available, one from a 0D detector (CH000) and 2 from the Measurements ROIs of a 1D detector. In that case the CH000 data has been selected and will therefore be use as feedback for the Adaptive algorithm.

Loss

All the Adaptive options are called *Loss* on the Scanner UI. These influence the adaptive algorithm, using previously probed positions and their feedback to guess the next point to probe. See the [Adaptive documentation](#) on *loss* to understand all the possibilities.

Navigator

From version 1.4.0, a new module has been added: the Navigator (`daq_utils.plotting.navigator`). It is most useful when dealing with 2D scans such as XY cartography. As such, it is not displayed by default. It consists of a tree like structure displaying all currently saved 2D scans (in the current dataset) and a viewer where selected scans can be displayed at their respective locations. It can be displayed using the *Settings* menu, *Show Navigator* option. [Fig. 7.40](#) shows the DAQ_scan extension with activated Navigator and a few scans. This navigator can also be used as a *Scan Selector* viewer to quickly explore and select areas to scan on a 2D phase space.

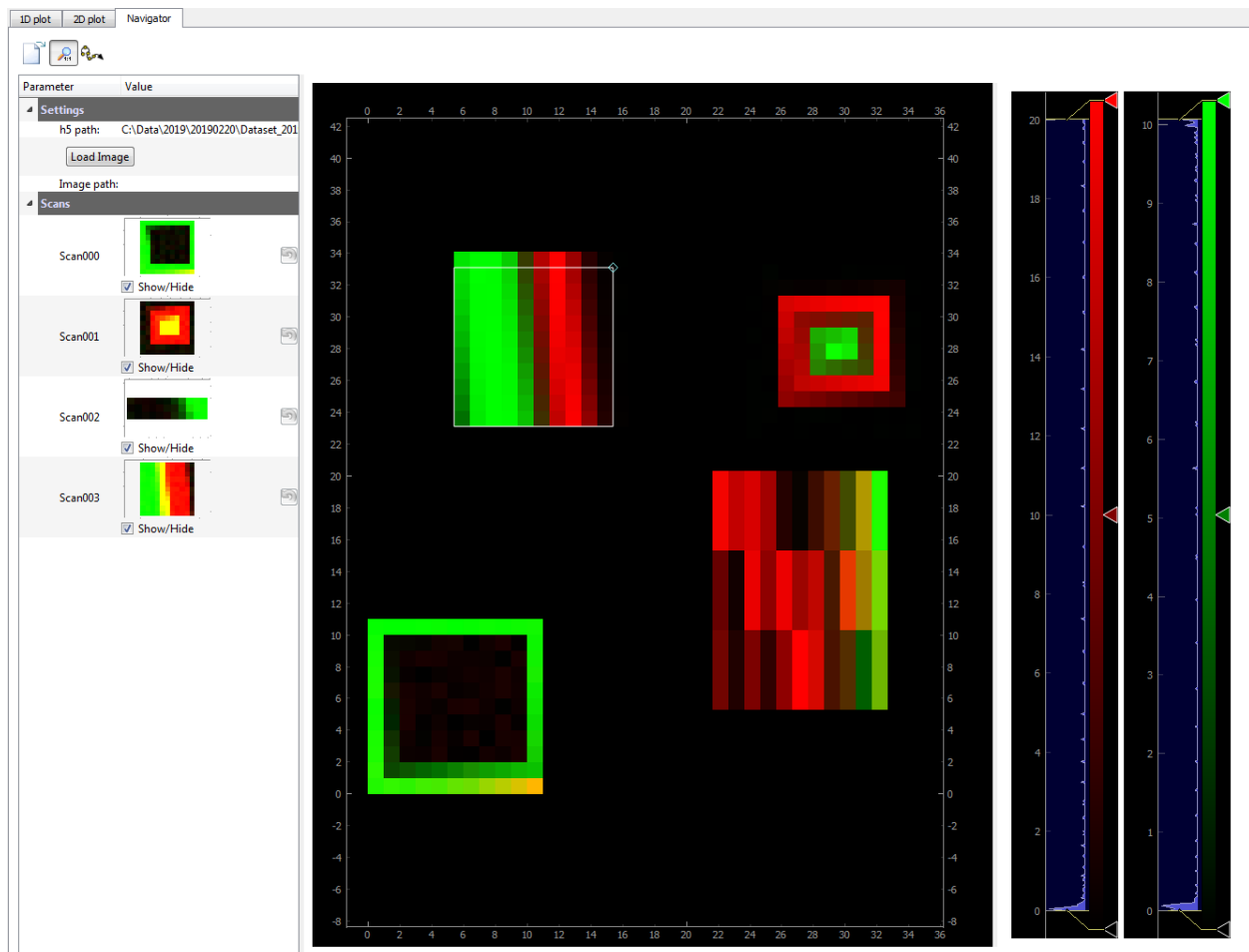


Fig. 7.40: An example of dataset displaying several 2D scans at their respective locations (up and right axis)

Scan Batch Manager

If the *Scan Batch Manager* is activated, a new menu entry will appear: *Batch Configs*, that let the user define, modify or load scan batch configurations. When loaded, a particular configuration will be displayed in the batch window. This window (see Fig. 7.41) displays (in a tree) a list of scans to perform. Each scan is defined by a set of actuators/detectors to use and scan settings (*Scan1D*, *Linear*... just as described in *Settings*).

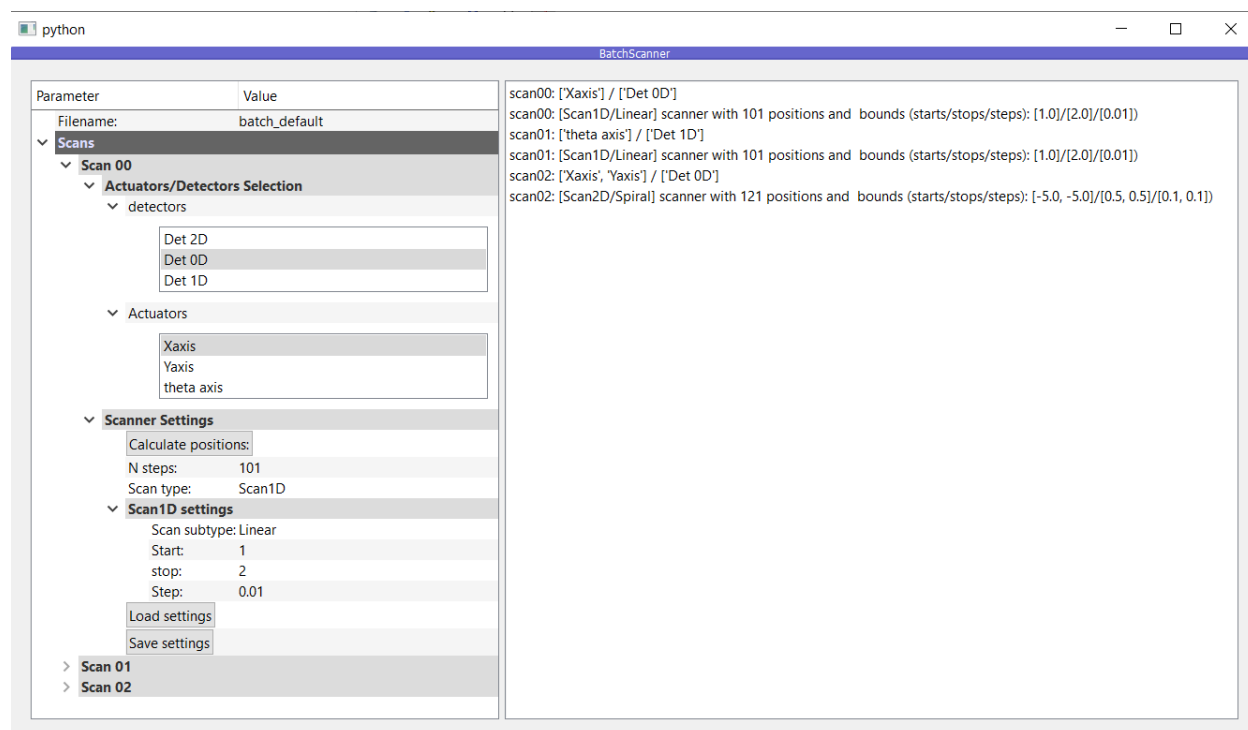



Fig. 7.41: An example of a Scan Batch configuration displaying several scans to perform

A new start button  will also appear on the main window to start the currently loaded scan batch.

DAQ Logger

This module is an extension of the dashboard, it will:

- ask you where to log data from all selected detectors
- save log datas in hierarchical binary files (compatible with the *H5Browser*)

The flow of this module is as follow:

- at startup you have to define/load a preset (see *Preset manager*) in the Dashboard
- Select DAQ_Logger in the actions menu
- Select the destination of the logged data: binary hdf5 file or SQL database

Introduction

In construction

Main Control Window

In construction

H5 saving

In construction

SQL Database saving

In construction

PID Module

Note: For now this module is **not compatible with PyMoDAQ 4**. Please use the PyMoDAQ 3.6.8 version, as mentioned latter in this documentation. We are currently working on to update the PID extension.

Introduction

This documentation is complementary to the video on the module :

<https://www.youtube.com/watch?v=u8ifY4WqQEA>

The PID module is useful if you would like to control a parameter of a physical system (a temperature, the length of an interferometer, the beam pointing of a laser...). In order to achieve this, you need a set of detectors to read the current state of the system, an interpretation of this reading, and a set of actuators to perform the correction.

Note: Notice that the speed of the corrections that can be achieved with this module are inherently limited below 100 Hz, because the feedback system uses a computer. If you need a faster correction, you should probably consider an analogic solution.

First example: a boiler

Let consider this physical system. Some water is put in a jar, let say we want to keep the temperature of the water to 40°C, this is our **setpoint**. The system is composed of a heating element (an actuator), and a thermometer (a detector).

The control of the heater and the thermometer is a prerequisite to achieve the control of the temperature, but we also need a logic. For example:

- if $T - T_{\text{setpoint}} < 5^{\circ}\text{C}$ then heater is ON

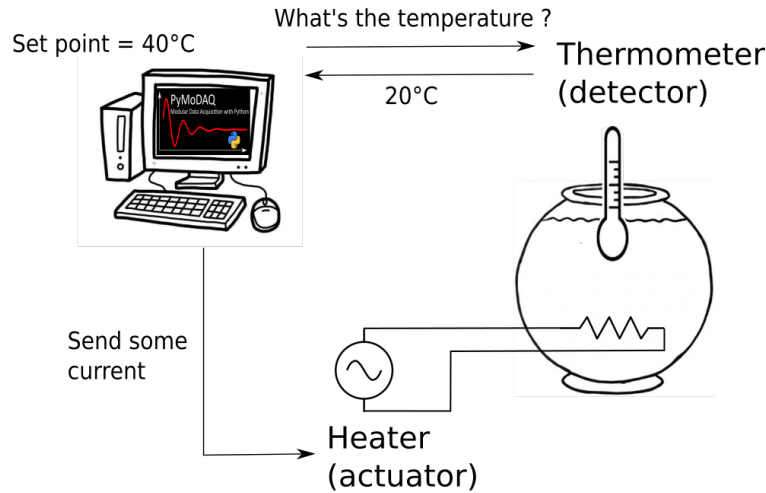


Fig. 7.42: The boiler system.

- if $T - T_{\text{setpoint}} > 5^{\circ}\text{C}$ then heater is OFF

With this logic, when the hot water will have dissipated enough energy in its environment to reach 35°C , the heater will be switch on to heat it up to 45°C and then switch off. The temperature of the water will then be oscillating approximately around 40°C .

The difference between the setpoint and the current value of the control parameter, here $T - T_{\text{setpoint}}$, is called the **error signal**.

The PID Model

Depending on the system you want to control, there will be a different number of actuators or detectors, and a different logic. For example, if you want to control the pointing of a laser on a camera, you will need a motorized optical mount to hold a mirror with two actuators that control the tip and tilt axes, what we call a **beam steering system**. The way you calculate your error signal will be different: you will need a way to define the center of the laser beam on the camera, like the barycenter of the illuminated pixels, and the error signal will be a 2D vector, one for the vertical and one for the horizontal direction.

Fig. 7.43: The beam steering scheme.

Another exemple consists it propagating a continuous laser in the two arms of an interferometer to produce an interference pattern. The phase of the fringes depending on the difference in the arms' lengths, it is possible to retrieve an error signal from this interference pattern to lock the interferometer, or even to sweep its length while it is locked.

The PID Model is a configuration of the PID module which depends on the physical system we want to control. It contains:

- the number and the dimensionality of the required detectors
- the number of actuators

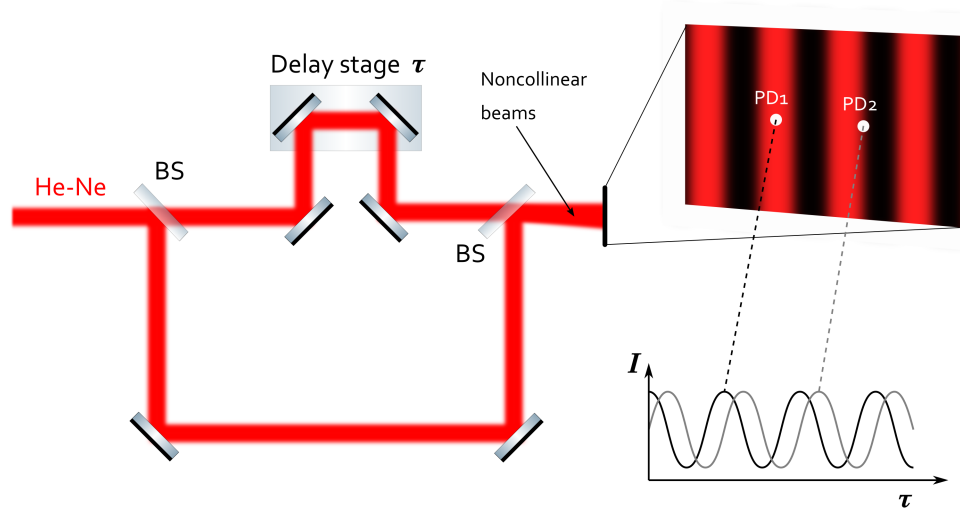


Fig. 7.44: The interferometer scheme.

- the number of setpoints
- the logic to calculate the error signal from the detectors' signals

A PID model is associated to each different physical system we want to control.

Demonstration with a virtual beam steering system

Lucky you, you do not need a real system to test the PID module! A computer and an internet connection are enough. For our demonstration, we will install some mock plugins that simulate a beam steering system.

Let us start from scratch, we follow the installation procedure of PyMoDAQ that you can find in the installation page: <https://pymodaq.cnrs.fr/en/latest/usage/Installation.html>

We suppose that you have Miniconda3 or Anaconda3 installed.

In a console, first create a dedicated environment and activate it

```
conda create -n mock_beam_steering python=3.8
```

```
conda activate mock_beam_steering
```

Install PyMoDAQ with the version that have been tested while writing this documentation

```
pip install pymodaq==3.6.8
```

and the Qt5 backend

```
pip install pyqt5
```

We also need to install (from source) another package that contains all the mock plugins to test the PID module. This step is optional if you wish to use the PID module with real actuators and detectors.

```
pip install git+https://github.com/PyMoDAQ/pymodaq_plugins_pid.git
```

Preset configuration

Launch a dashboard

dashboard

Note: If at this step you get an error from the console, try to update to a newest version of the package “tables”, for instance `pip install tables==3.7` and try again to launch a dashboard.

In the main menu go to

Preset Modes > New Preset

Let us choose a name, for example **preset_mock_beam_steering**.

Under the Moves section add two actuators by selecting **BeamSteering** in the menu, and configure them as follow. The **controller ID** parameter could be different from the picture in your case. Let this number unchanged for the first actuator, but it is important that all the two actuators and the detector have the same controller ID number. It is also important that the controller status of the first actuator be **Master**, and that the status of the second actuator and the detector be **Slave**. (This configuration is specific to the demonstration. Underneath the actuators and the detector share a same virtual controller to mimic a real beam steering system, but you do not need to understand that for now!)

Now, add a 2D detector by selecting **DAQ2D/BeamSteering** in the menu, and configure it as follow

and click **SAVE**.

Back to the dashboard menu

Preset Modes > Load preset > preset_mock_beam_steering

Your dashboard should look like this once you have grabbed the camera and unwrapped the option menus of the actuators.

If you now try a relative move with Xpiezo or Ypiezo, you will see that the position of the laser spot on your virtual camera is moving horizontally or vertically, as if you were playing with a motorized optical mount.

Our mock system is now fully configured, we are ready for the PID module!

PID module

The loading of the PID module is done through the dashboard menu

Extensions > PID Module

It will popup a new window, in Model class select `PIDModelBeamSteering` and **(1) initialize the model**.

Configure it as follow:

- camera refresh time (in the dashboard) = 200 ms
- PID controls/sample time = 200 ms
- PID controls/refresh plot time = 200 ms
- threshold = 2

Fill in information about this manager

Parameter	Value
Filename:	preset_mock_pid_beam_steering
Use PID as actuator:	<input type="checkbox"/>
Moves:	
Actuator 00	
Name:	Xpiezo
Init?:	<input checked="" type="checkbox"/>
Settings:	
Main Settings:	
Actuator type:	BeamSteering
Controller ID:	5166
TCP/IP options:	
Actuator Settings:	
MultiAxes:	
is Multiaxes:	<input checked="" type="checkbox"/>
Status:	Master
Axis:	H
Units:	
Epsilon:	1
Timeout (s):	10
Bounds:	
Scaling:	
Actuator 01	
Name:	Ypiezo
Init?:	<input checked="" type="checkbox"/>
Settings:	
Main Settings:	
Actuator type:	BeamSteering
Controller ID:	5166
TCP/IP options:	
Actuator Settings:	
MultiAxes:	
is Multiaxes:	<input checked="" type="checkbox"/>
Status:	Slave
Axis:	V
Units:	
Epsilon:	1
Timeout (s):	10
Bounds:	
Scaling:	
Detectors:	
Det 00	

Add +

Cancel Save

Fig. 7.45: The mock actuators configuration.



Fig. 7.46: The mock camera configuration.

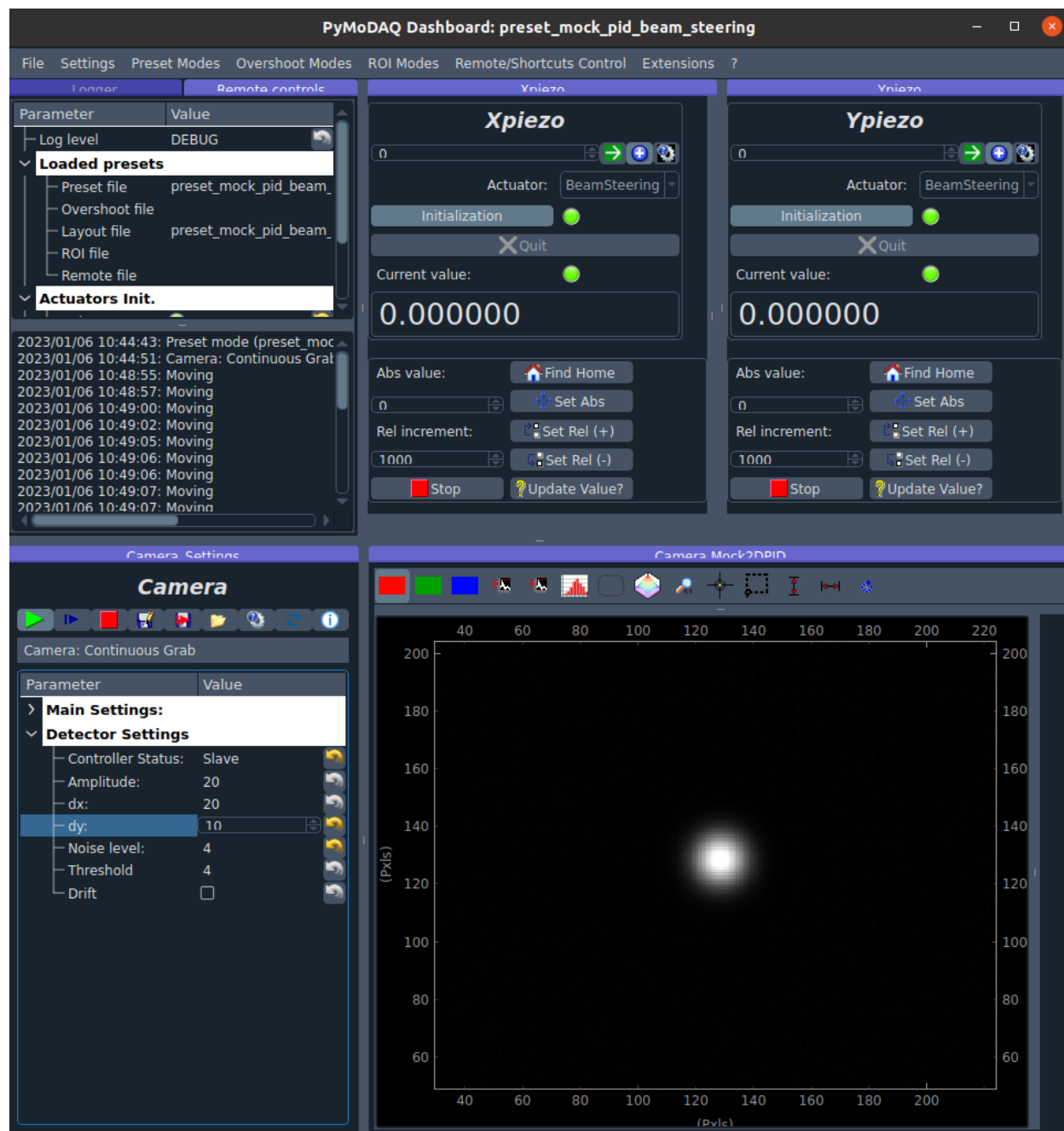


Fig. 7.47: The dashboard after loading the preset.

Then (2) **initialize the PID** and (3) **start the PID** loop with the **PLAY** button. Notice that at this stage the corrections are calculated, but the piezo motors are not moving. It is only when you will (4) **untick the PAUSE** button that the corrections will be applied.

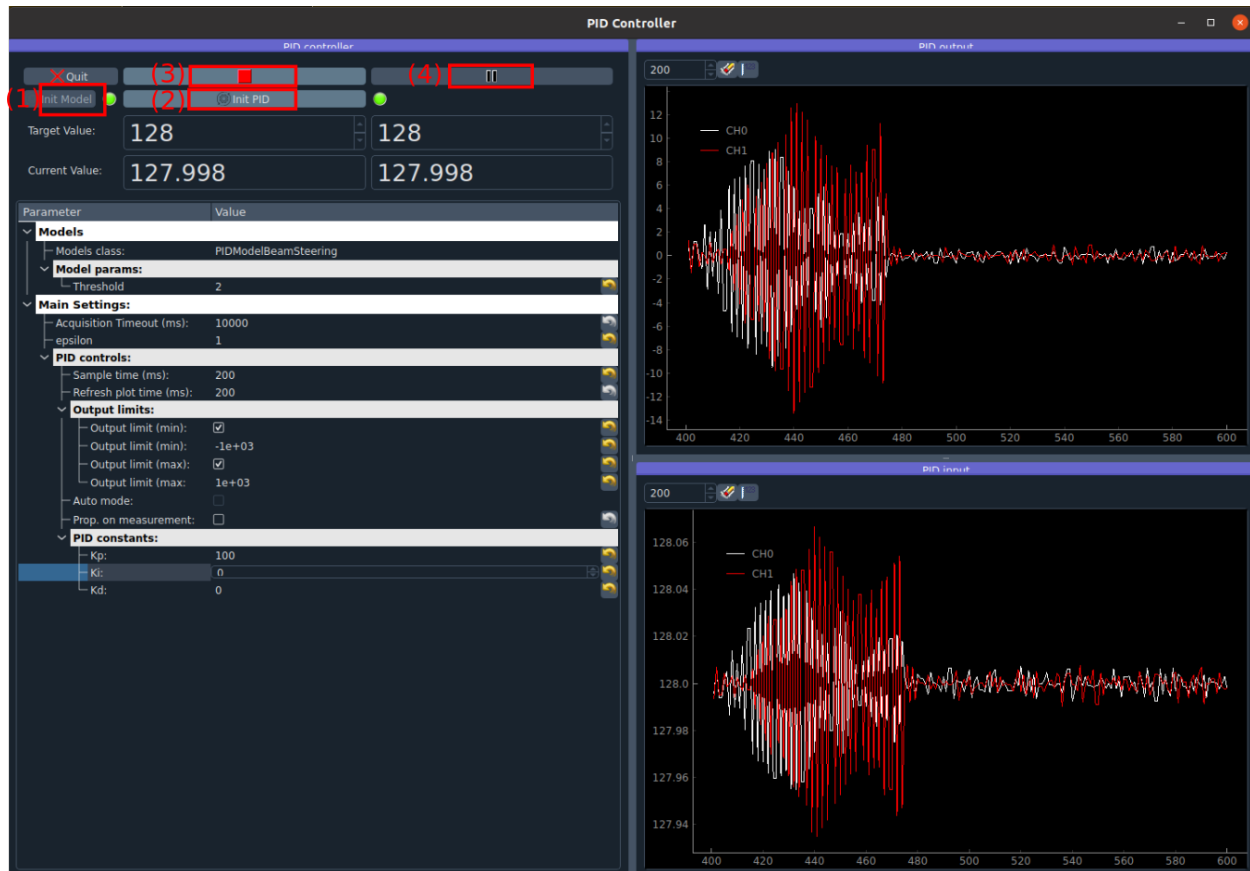


Fig. 7.48: The PID module interface.

PID configuration

Output limits

The output limits are here mainly to prevent the feedback system to send crazy high corrections and move our beam out of the chip.

If we put them too low, the feedback system will only send tiny corrections, and it will take a long time to correct an error, or if we change the setpoint.

If we increase them, then our system will be able to move much faster.

The units of the output limits are the same as the piezo motors, let say in microns. Put an output limit to +500 means “If at any time the PID outputs a correction superior to 500 microns, then only correct 500 microns.”

The output limits are not here to slow down the correction, if we want to do that we can decrease the proportional parameter (see next section). They are here to define what we consider as a crazy correction.

To define them we can pause the PID loop and play manually with the piezo actuators. We can see that if we do a 10000 step, we almost get out of the chip of the camera, thus an **output limit of 1000** seems reasonable.

If we do a big change of setpoint and see that every step of the piezo corresponds to the output limit we configured, then it means the corrections are saturated by the output limits.

Configuring the PID parameters

The proportional, integral, derivative parameters of the PID filter, respectively K_p , K_i and K_d , will dictate the behavior of the feedback system.

Stay at a fixed position while the correction loop is closed, and start with $K_p = 1$, $K_i = 0$, $K_d = 0$. Then change the setpoint to go close to an edge of the camera. We see that the system is doing what it is supposed to do: the beam goes to the setpoint... but veeeeeeeeeeery slowly. This is not necessarily bad. If your application does only need to keep the beam at a definite position (e.g. if you inject an optical fiber), this can be a good configuration. If we take a look at the **PID input** display, which is just the measured position of the beam on the chip in pixel, we can see that reducing K_p will decrease the fluctuations of the beam around the target position. Thus a low K_p can increase the stability of your pointing.

Let say now that we intend to move regularly the setpoint. We need a more reactive system. Let us increase progressively the value of K_p until we see that the beam start to oscillate strongly around the target position (this should happen for K_p close to 200 - 300). We call this value of K_p the ultimate gain. Some heuristic method says that dividing the ultimate gain by 2 is a reasonable value for K_p . So let us take **$K_p = 100$** .

We will not go further in this documentation about how to configure a PID filter. For lots of applications, having just K_p is enough. If you want to go further you can start with this Wikipedia page: https://en.wikipedia.org/wiki/PID_controller.

Automatic control of the setpoints

Let us imagine now that we want to use this beam to characterize a sample, and that we need a long acquisition time at each position of the beam on the sample to perform our measurement. Up to now our feedback system allows to keep a stable position on the sample, which is nice. But it would be even better to be able to scan the surface of the sample automatically rather than moving the setpoints manually. That is the purpose of this section!

In order to do that, we will create virtual actuators on the dashboard that will control the setpoints of the PID module. Then, PyMoDAQ will see them as standard actuators, which means that we will be able to use any of the other modules, and in particular, perform any scan that can be configured with the DAQ_Scan module.

Preset configuration

Start with a fresh dashboard, we have to change a bit the configuration of our preset to configure this functionality. Go to

Preset Modes > Modify preset

and select the one that we defined previously. You just need to tick **Use PID as actuator** and save it.

Moving the setpoints from the dashboard

Load this new preset. Notice that it now automatically loaded the PID module, and that our dashboard got two more actuators of type PID named **Xaxis** and **Yaxis**. If you change manually the position of those actuators, you should see that they control the setpoints of the PID module.

Moving the setpoints with the DAQ Scan module

Those virtual actuators can be manipulated as normal actuators, and you can ask PyMoDAQ to perform a scan of those guys! Go to

Extensions > Do scans

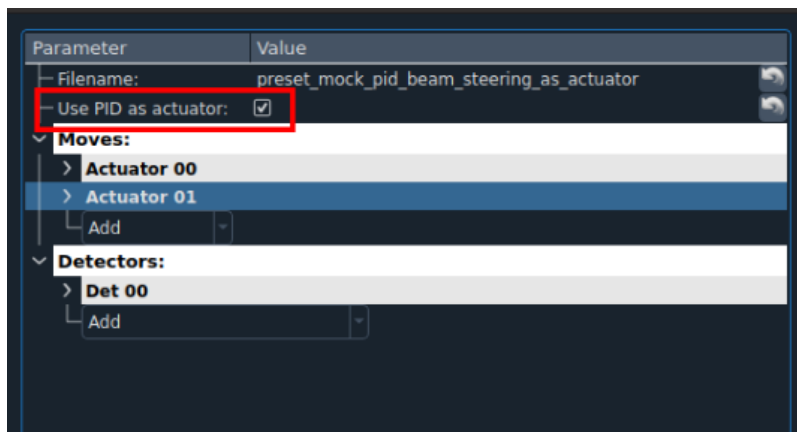


Fig. 7.49: Configuration of the preset for automatic control of the setpoints.

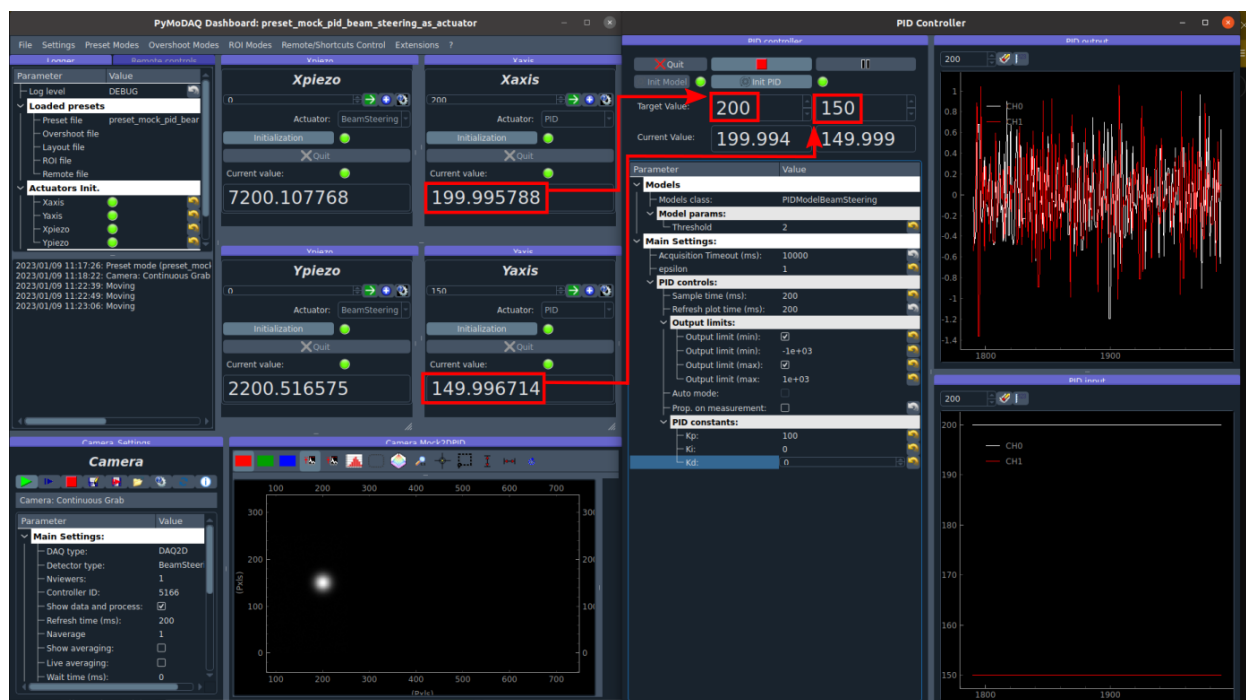


Fig. 7.50: Virtual actuators on the dashboard control the setpoints of the PID module.

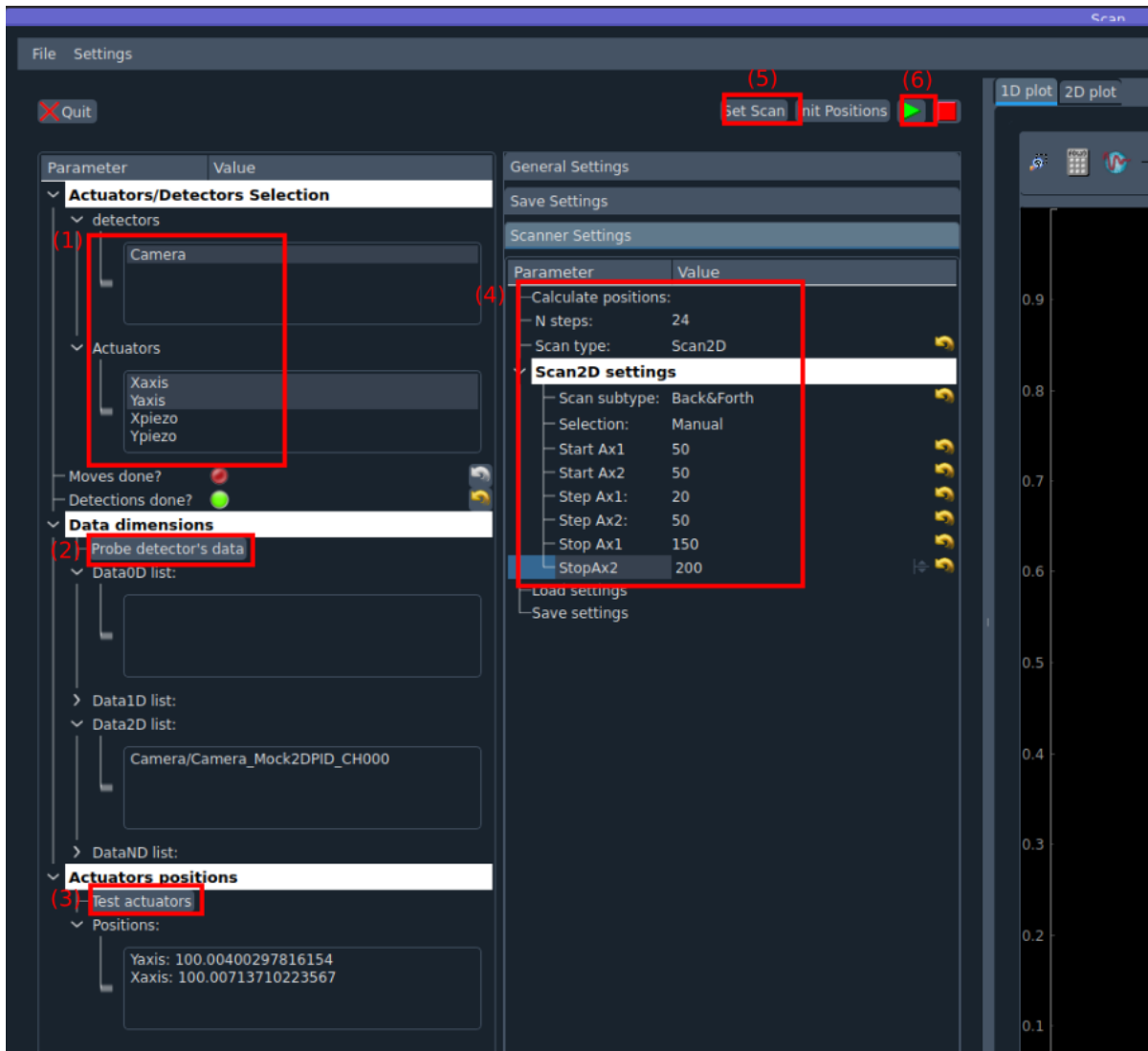


Fig. 7.51: Configuration of a scan with the DAQ_Scan module.

Some popup windows will ask you to name your scan. This is not important here. Configure the scan as follow

- (1) Select **Camera**, **Xaxis**, **Yaxis** (maintain Ctrl command to select several actuators)
- (2) Click **Probe detector's data**
- (3) Click **Test actuators** and select a position at the center of the camera
- (4) **Define a 2D scan** as follow. Notice that Ax1 (associated to the Xaxis) corresponds to the main loop of the scan: its value is changed, then all the values of Ax2 are scanned, then the value of Ax1 is changed, and so on...
- (5) **Set scan**
- (6) **Start** and look at the camera

The beam should follow automatically the scan that we have defined. Of course in this demonstration with a virtual system, this sounds quite artificial, but if you need to perform stabilized scans with long acquisition times, this feature can be very useful!

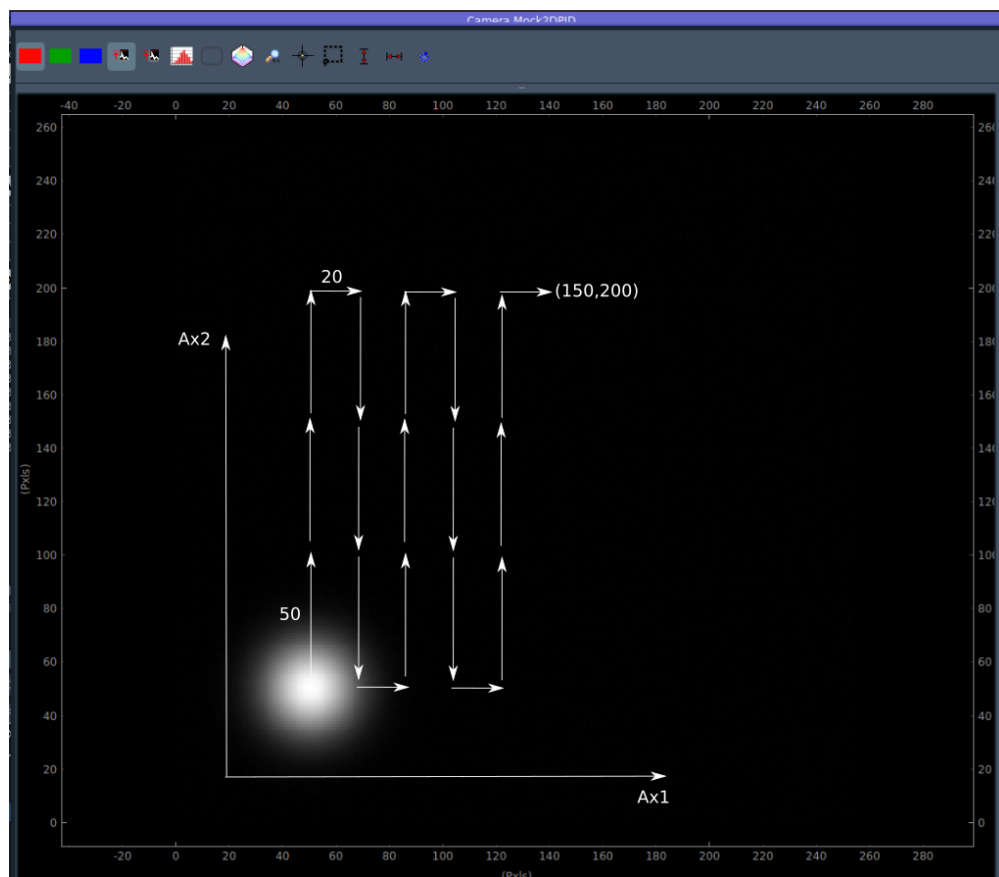


Fig. 7.52: Movement of the beam on the camera with a scan of the setpoints.

How to write my own PID application?

Package structure for a PID application

To write your own PID application, you should create a package with a similar structure as a standard *py-modaq_plugins_xxx* package. There are few modifications. Let us have a look at the *pymodaq_plugins_pid*.

Notice there is a *models* folder next to the *hardware* folder, at the root of the package. This folder will contains the PID models.

Then python will be able to probe into those as they have been configured as entry points during installation of the package. You should check that those lines are present in the *setup.py* file of your package.

This declaration allows PyMoDAQ to find the installed models when executing the PID module. Internally, it will call the *get_models* method that is defined in the *daq_utils*.

In order to use the PID module for our specific physical system, we need:

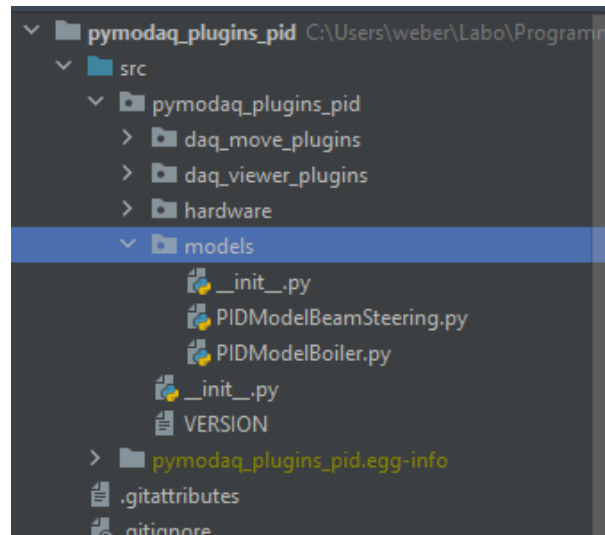
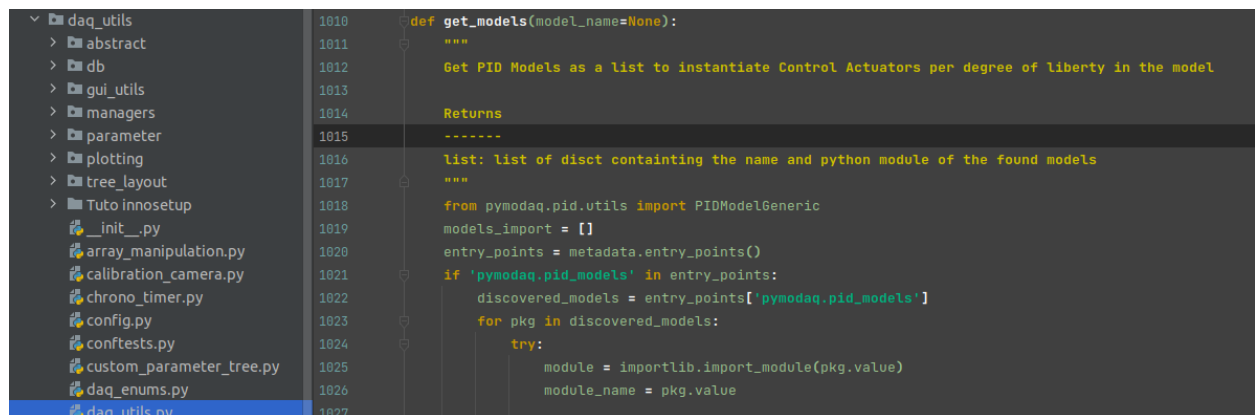


Fig. 7.53: Structure of a package containing PID models.

Fig. 7.54: Declaration of entry points in the setup.py file.

Fig. 7.55: The *get_models* method in the *daq_utils*.

- A **set of detector and actuator plugins** that is needed to stabilize our system.
- A **PID model** to implement the logic that is needed to translate the detectors' signals into a correction.

Detector/actuator plugins

In the beam steering example, this corresponds to one actuator plugin (if you use the same motor model for horizontal and vertical axis), and a camera plugin.

The first thing to do is to check the [list of readily available plugins](#).

The easy scenario is that you found that the plugins for your hardware are already developed. You then just have to test if they properly move or make an acquisition with the [DAQ Move](#) and [DAQ Viewer](#) modules. That's it!

If there is no plugin developed for the hardware you want to use, you will have to develop your own. Don't panic, that's quite simple! Everything is explained in the [Plugins](#) section of the documentation, and in [this video](#). Moreover, you can find a lot of examples for any kind of plugins in the list given above and in the [GitHub repository of PyMoDAQ](#). If at some point, you stick on a problem, do not hesitate to raise an issue in the GitHub repository or address your question to the mailing list pymodaq@services.cnrs.fr.

Note: It is not necessary that the plugins you use are declared in the same package as your model. Actually a model is in principle independent of the hardware. If you use plugins that are declared in other packages, you just need them to be installed in your python environment.

How to write a PID model?

Naming convention

Similarly to [plugins](#), there exist naming conventions that you should follow, so that PyMoDAQ will be able to parse correctly and find the classes and the files that are involved.

- The name of the file declaring the PID model should be named **PIDModelXxxx.py**
- The class declared in the file should be named **PIDModelXxxx**

Number of setpoints and naming of the control modules

The number of setpoints, their names, and the naming of the control modules are declared at the beginning of the class declaration. It is important that **those names are reported in the preset file associated to the model**. We understand now that those names are actually set in the PID model class.

The required methods of a PID model class

There are two required methods in a PID model class:

- **convert_input** that will translate the acquisitions of the detectors into an understandable input for the PID filter (which is defined in an external package).
- **convert_output** that will translate the output of the PID filter(s) into an understandable order for the actuators.

In this example of the `PIDModelBeamSteering`, the `convert_input` method get the acquisition of the camera, remove the *threshold* value defined by the user through the UI (this is to remove the background noise), calculate the center of mass of the image, and send the coordinates as input to the PID filter.

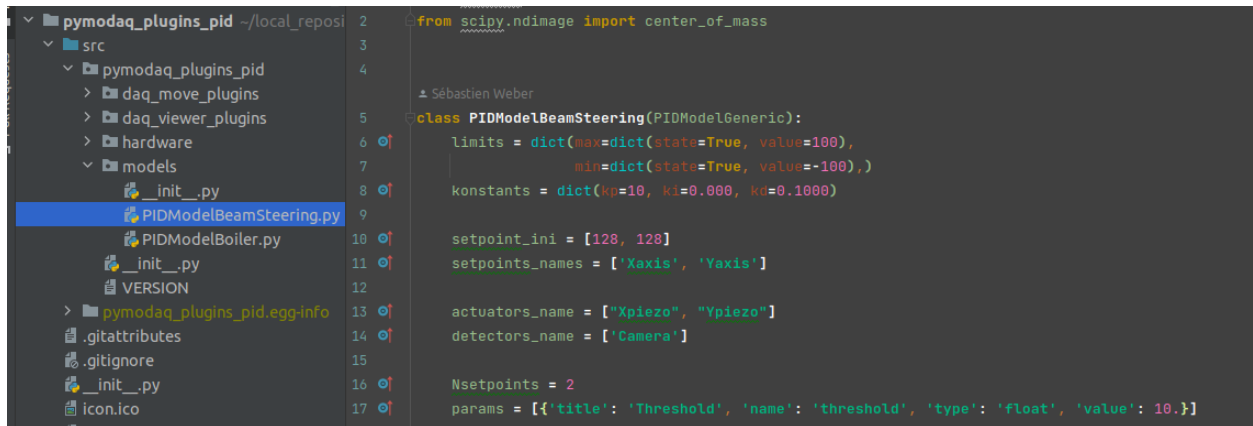


Fig. 7.56: Configuration of a PID model.

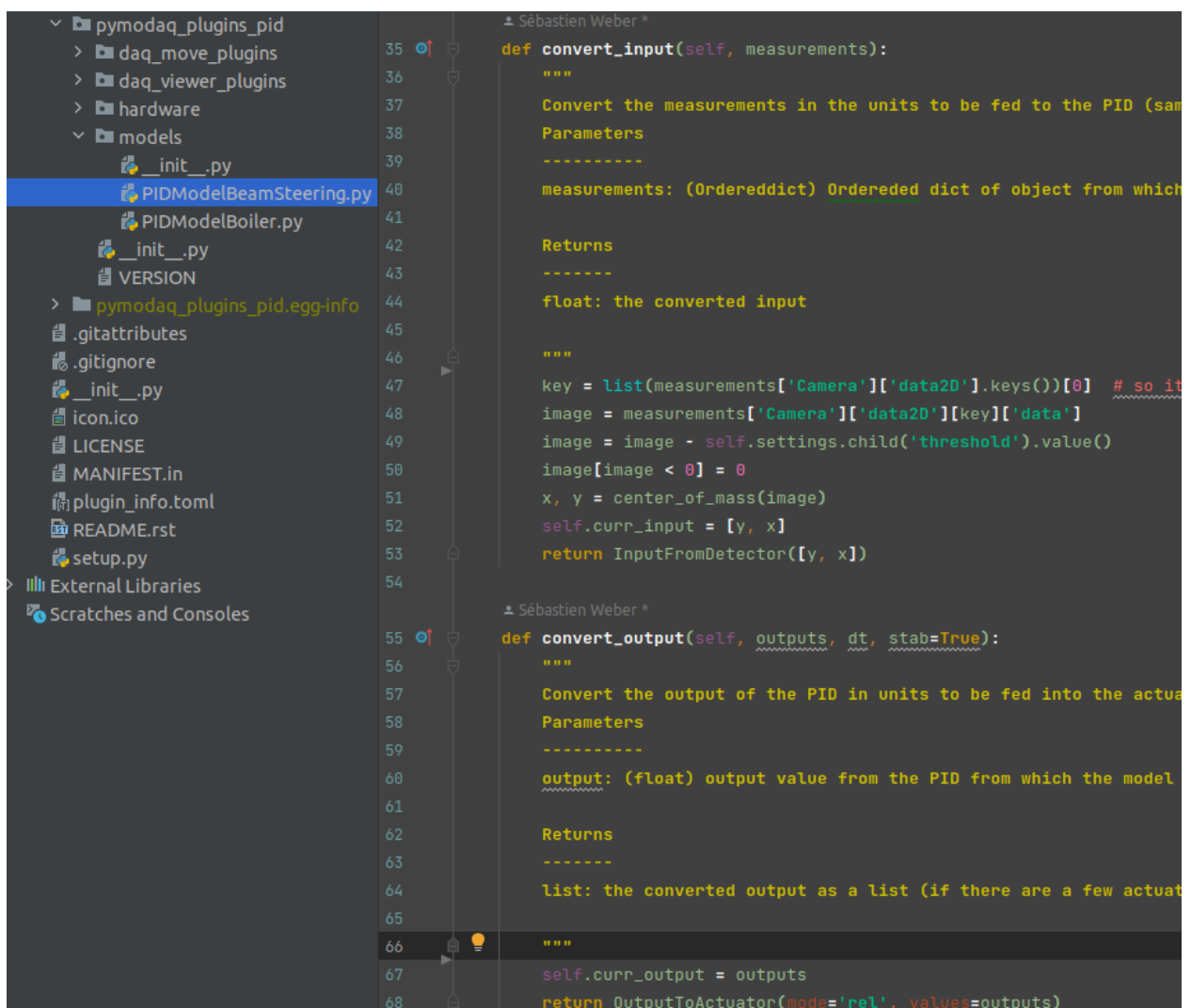


Fig. 7.57: The important methods of a PID model class.

Note: The PID filter is aware of the setpoints values, thus you just have to send him *absolute* values for the positioning of the system. He will calculate the difference himself.

As for the *convert_output* method, it only transfers the output of the PID filter directly as *relative* orders to the actuators.

Note: The output of the PID filter is a correction that is *relative* to the current values of the actuators.

That's it!

Note: In this example, there is actually no other methods defined in the model, but you can imagine more complex systems where, for example, the translation from the detectors acquisitions to the input to the filter would need a calibration scan. Then you will probably need to define other methods. But, whatever it is, all the logic that is specific to your system should be defined in this class.

If you want to go deeper, the next section is for you!

PID module internals

This section is intended for the advanced user that intend to develop its custom application based on the PID module, or the one that is simply curious about the PID module internals. We will try to introduce here the main structure of the module, hoping that it will help to grasp the code more easily :)

Files locations

The files regarding the PID module are stored in the `/src/pymodaq/pid/` folder which contains:

- **utils.py** which defines some utility classes, and in particular the **PIDModelGeneric** class from which all PID models inherit.
- **daq_move_PID.py** which defines a virtual actuator that control the setpoint of the PID module. This is useful for example if the user wants to scan the control parameter while it is locked.
- **pid_controller.py**. It is the core file of the module that defines the **DAQ_PID** and the **PIDRunner** classes that will be presented below.

Packages

- **PyMoDAQ/pymodaq_plugins_pid** This package contains some mock plugins and models to test the module without hardware. It is for development purposes and thus optional.
- **PyMoDAQ/pymodaq_pid_models** This package stores the PID models that have already been developped. Better to have a look before developping its own!

General structure of the module

Fig. 7.58: The structure of the PID module.

The **DAQ_PID** class is the main central class of the module. It manages the initialization of the program: settings of the user interface, loading of the PID model, instantiation of the **PIDRunner** class... It also makes a bridge between the user, who acts through the UI, and the **PIDRunner** class, which is the one that is in direct relation with the detectors and the actuators.

Since each of those classes is embedded in a thread, the communication between them is done through the **command_pid_signal** and the **queue_command** method.

The **PIDRunner** class is created and configured by the **DAQ_PID** at the initialization of the PID loop. It is in charge of synchronizing the instruments to perform the PID loop.

A **PIDModel** class is defined for each physical system the user wants to control. Here are defined the actuator/detector modules involved, the number of setpoints, and the methods to convert the information received from the detectors as orders to the actuators to perform the desired control.

The PID loop

The conductor of the PID loop is the **PIDRunner**, in particular the **start_PID** method. The workflow for each iteration of the loop can be mapped as in the following figure.

Fig. 7.59: An iteration of the PID loop.

The starting of the PID loop is triggered by the user through the **PLAY button**.

The **PIDRunner** will ask the detector(s) to start an acquisition. When all are done, the **wait_for_det_done** method will send the data (**det_done_datas**) to the **PIDModel** class.

A **PIDModel** class should be defined for each specific physical system the user wants to control. Here are defined how much detectors/actuators are involved, and how the information sent by the detector(s) should be converted as orders to the actuators (**output_to_actuators**) to reach the targeted position (the setpoint). The **PIDModel** class is thus an interface between the **PID** class and the detectors/actuators. The important methods of those classes are **convert_input**, which will convert the detectors data to an input for the PID object, and the **convert_output** method which will translate the output of the PID object to the actuators.

The **PID** class is defined in an external package (simple_pid: <https://github.com/m-lundberg/simple-pid>). It implements a pid filter. The tunnings (Kp, Ki, Kd) and the setpoint are configured by the user through the user interface. From the input, which corresponds to the current position of the system measured by the detectors, it will return an output that corresponds to the order to send to the actuators to stabilize the system around the setpoint (given that the configuration has been done correctly). Notice that the input for the PID object should be an *absolute* value, and not a relative value from the setpoint. The setpoint is entered as a parameter of the object so it can make the difference itself.

Bayesian Optimisation

First of all, this work is heavily supported by the work of Fernando Nogueira through its python package: [bayesian-optimization](#) and the underlying use of Gaussian Process regression from [scikit-learn](#).

Introduction

You'll find below, a very short introduction, for a more detailed one, you can also read [this article](#) from Okan Yenigun from which this introduction is derived.

Bayesian optimization is a technique used for the global optimization (finding an optimum) of black-box functions. Black box functions are mathematical functions whose internal details are unknown. However given a set of input parameters, one can evaluate the possibly noisy output of the function. In the PyMoDAQ ecosystem, such a black box would often be the physical system of study and the physical observation we want to optimize given a certain number of parameters. Two approaches are possible: do a grid search or random search using the DAQ_Scan extension that can prove inefficient (you can miss the right points) and very lengthy in time or do a more intelligent phase space search by building a probabilistic surrogate model of our black box by using the history of tested parameters.

Gaussian Processes:

The surrogate model we use here is called Gaussian Process, GP. A Gaussian process defines a distribution of functions potentially fitting the data. This finite set of function values follows a multivariate Gaussian distribution. In the context of Bayesian optimization, a GP is used to model the unknown objective function, and it provides a posterior distribution over the function values given the observed data.

From this distribution, a mean value (μ) and standard deviation (std) of the function distribution is computed. These are then used to model our black box system. To go one step beyond, the algorithm should predict which parameters should be probed next to optimize the mean and std. For this we'll construct a simple function based on the probability output of the GP: the acquisition function.

Note: GPs are themselves based on various kernels (or covariance matrix or *function generator*). Which kernel to use may depend on your particular problem, even if the standard ones (as provided in PyMoDAQ) should just work fine. If you want to know more on this just browse this [thesis](#).

Acquisition function:

Choosing which point to probe next is the essential step in optimizing our black box. It should quantify the utility of the next point either to directly optimize our problem or to increase the fitness of the model. Should it favor the exploration of the input parameters phase space? Should it perform exploitation of the known points to find the optimum?

All acquisition function will allow one or the other of these, or propose an hyperparameter to change the behaviour during the process of optimisation. Among the possibilities, you'll find:

- The Expected Improvement function (EI)
- The Upper Confidence Bound function (UCB)
- The Probability of Improvement function (PI)
- ...

You can find details and implementation of each in [here](#). PyMoDAQ uses by default the Upper Confidence Bound function together with its *kappa* hyperparameter, see [Settings](#) and [here](#).

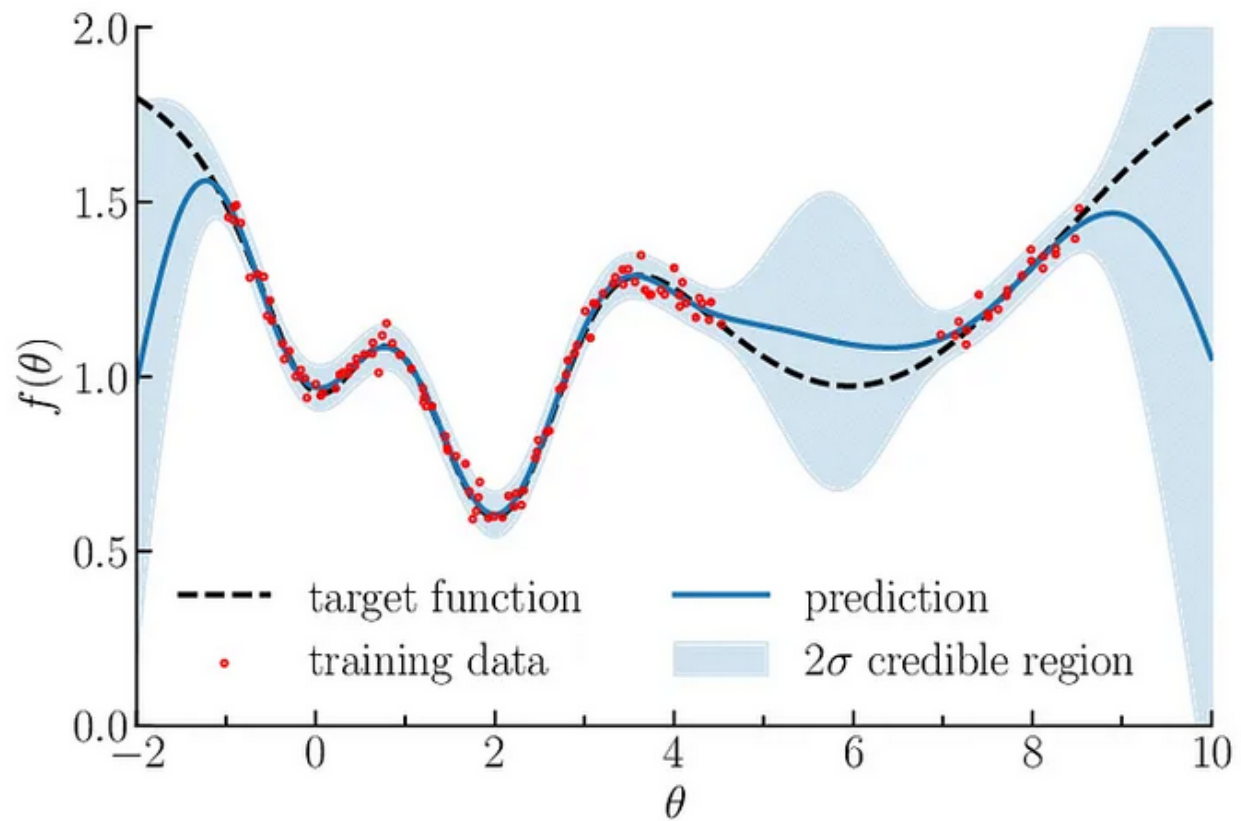


Fig. 7.60: Illustration of Gaussian process regression in one dimension. Gaussian processes are specified by an estimation function and the uncertainty function evolving constantly as more and more *points* are being tested. [Source](#)

Note: You can find a notebook illustrating the whole optimisation process on PyMoDAQ's Github: [here](#), where you can define your black box function (that in general you don't know) and play with kernels and utility functions.

Usage

Fig. 7.61 shows the GUI of the Bayesian Optimisation extension. It consists of three panels:

- Settings (left): allow configuration of the model, the search bounds, the acquisition function, selection of which detector and actuators will *participate* to the optimisation.
- Observable (middle): here will be plotted the evolution of the result of the optimisation. On the top, the best reached target value will be plotted. On the bottom, the coordinates (value) of the input parameters that gave the best reached target will be plotted.
- Probed Data: this is a live plotter of the history of tested input parameters and reached target

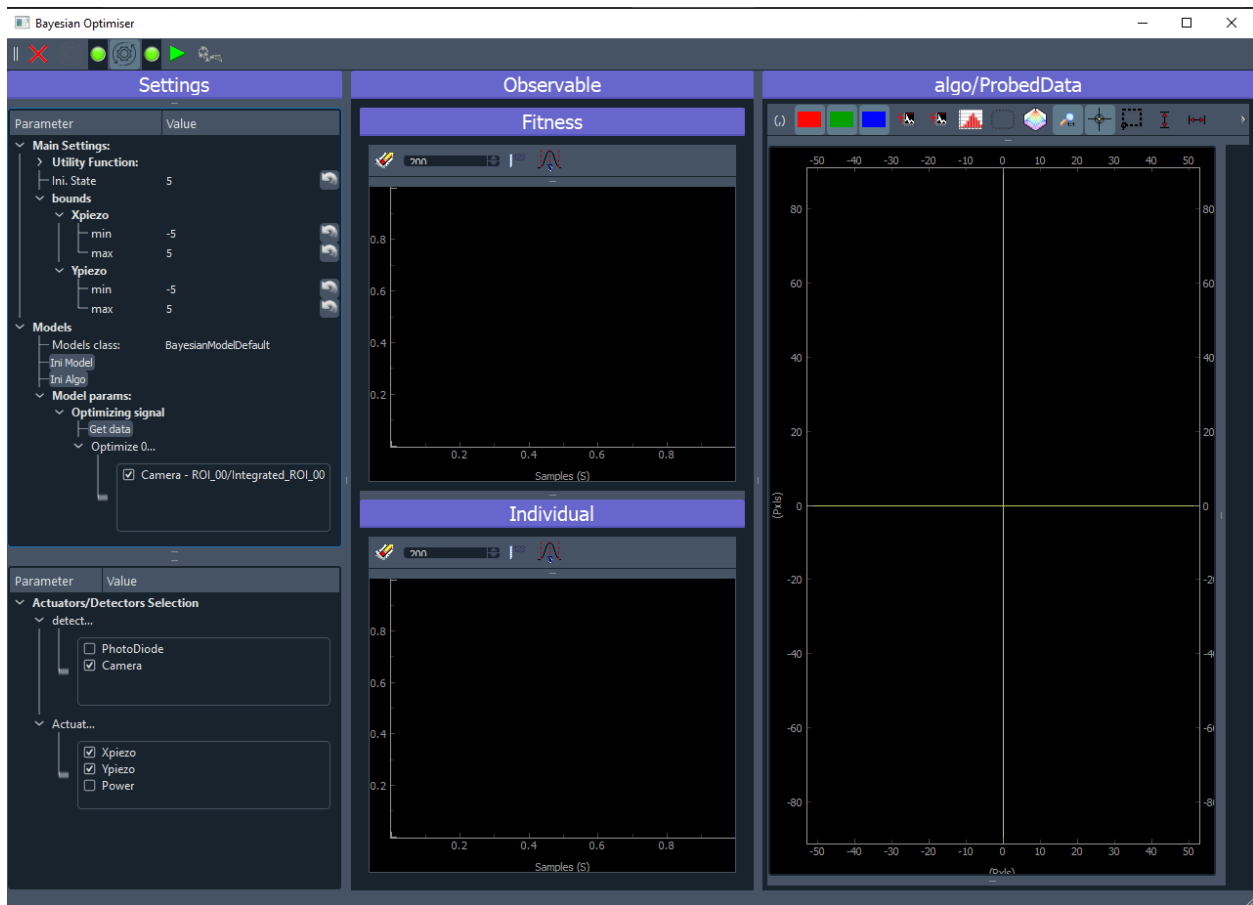







Fig. 7.61: User Interface of the Bayesian Optimization extension.

Toolbar:

- : quit the extension
- : Initialise the selected model
- : Initialise the Bayesian algorithm with given settings
- : Run the Bayesian algorithm
- : Move the selected actuators to the values given by the best target reached by the algorithm

Settings

1. Actuators and detectors

First of all, you'll have to select the detectors and actuators that will be used by the algorithm, see [Fig. 7.62](#).

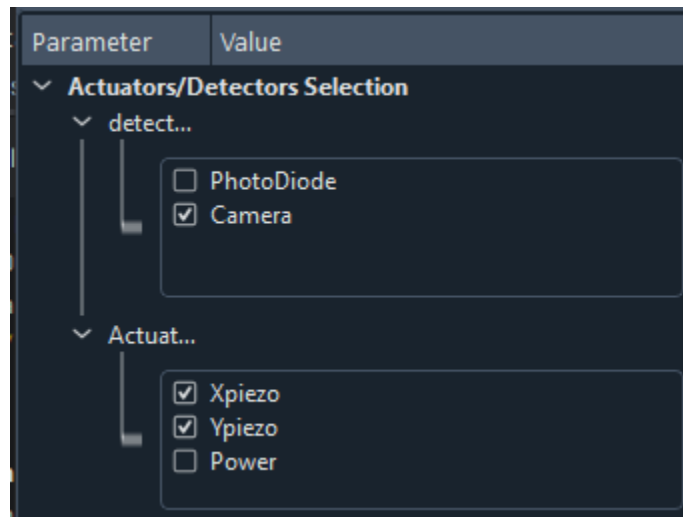


Fig. 7.62: Zoom on the settings of the GUI for selection of the detectors and actuators to be used in the optimization.

2. Model selection

Then you have to select a *model* (see [Fig. 7.63](#) and [Models](#)) allowing the customization of the extension with respect of what is the signal to be optimized, which particular plot should be added... . If the signal to be optimized is just one of the 0D data generated by one of the selected detector, then the *BayesianModelDefault* is enough and no model programming is needed. If not, read [Models](#). In the case of the *BayesianModelDefault*, you'll have to select a 0D signal to be used as the target to be optimized, see bottom of [Fig. 7.63](#).

3. Algorithm parameters

Then, you'll have to specify the number of initial random state. This number means that before running a fit using the GPs, the first N iteration will be made using a random choice of input parameters among the considered bounds. This allows for a better initial exploration of the algorithm.

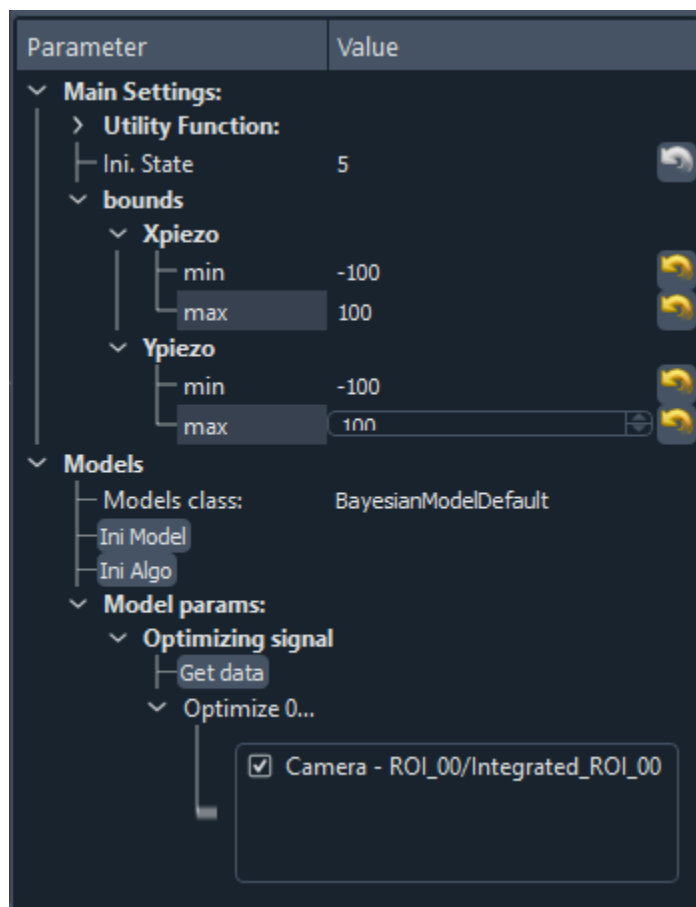


Fig. 7.63: Zoom on the settings of the GUI.

The value of the bounds is a crucial parameter. You have to enter the limits (min/max) for each selected actuator. The algorithm will then optimize the signal on this specified phase space.

Then you can run the algorithm,  button, and see what happens...

Note: Some parameters of the algorithm can be changed on the fly while the algorithm is running. This is the case for:

- the bounds
- the utility function hyper parameters

Observable and Probed Data

Once you run the algorithm, the plots will be updated at each loop. The observable will update the current best reached target value (fitness) and corresponding values of the actuators (input parameters). The right panel will plot all the collected targets at their respective actuators value. In the case of a 2D optimisation, it will look like on figure Fig. 7.64. The white crosshair shows the current tested target while the yellow crosshair shows the best reached value.

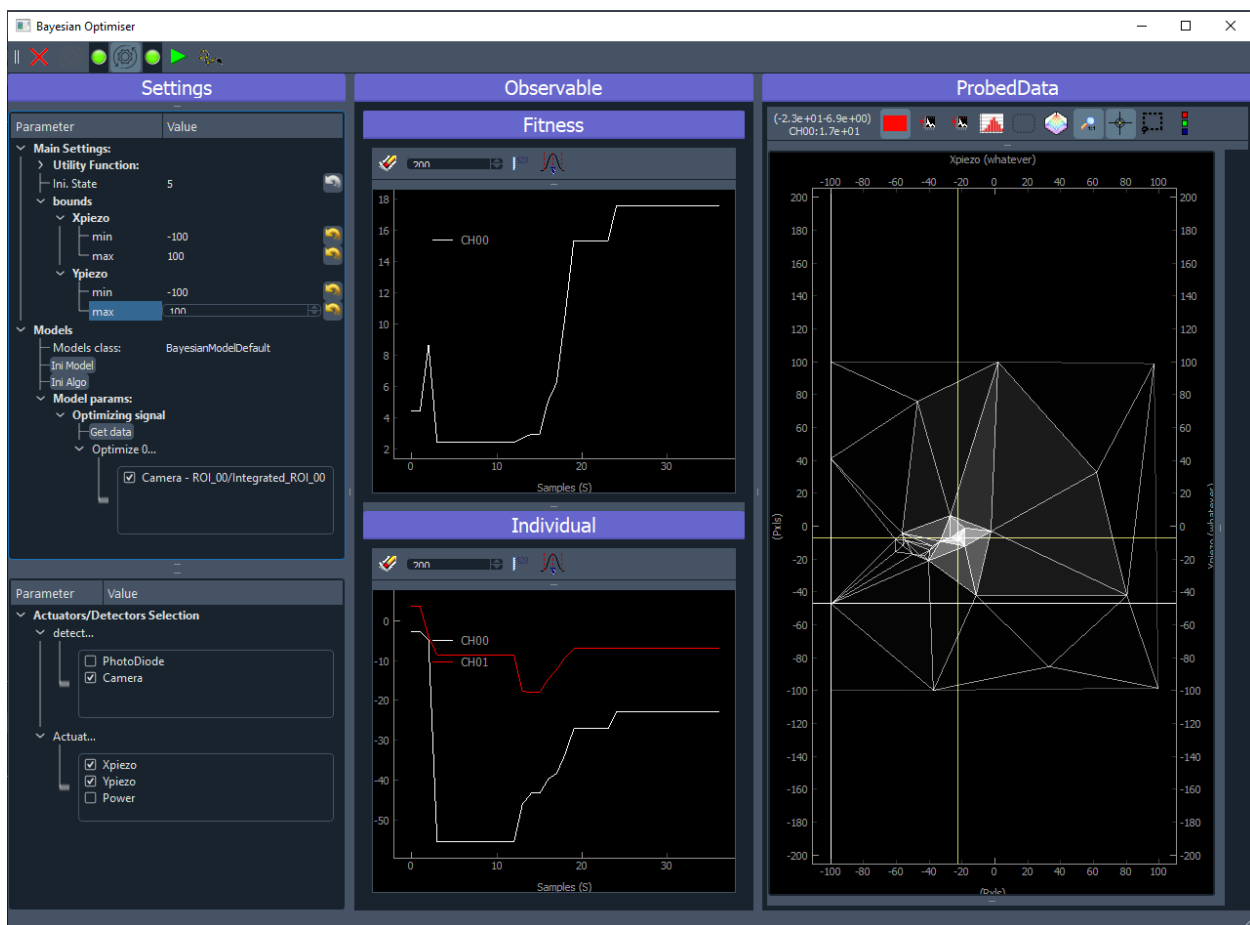





Fig. 7.64: User Interface of the Bayesian Optimization extension during a run.

Once you stop the algorithm (pause it in fact), the  button will be enabled allowing to move the actuators to the best reached target values ending the work of the algorithm. If you want you can also restart it. If you press the  button, the algorithm will begin where it stops just before. If you want to reinitialize it, then press the  button twice (eventually changing some parameters in between).

Models

In case the default model is not enough. It could be because what you want to optimize is a particular mathematical treatment of some data, or the interplay of different data (like the ratio of two regions of interest) or whatever complex thing you want to do.

In that case, you'll have to create a new Bayesian model. To do so, you'll have to:

1. create a python script
2. place it inside the models folder of a PyMoDAQ plugin (it could be a plugin you use with custom instruments, or you could devote a plugin just for holding your models: PID, Optimization... In that case, no need to publish it on pypi. Just hosting it locally (and backed up on github/gitlab) will do. You'll find an example of such a Model in the *pymodaq_plugins_mockexamples*
3. create a class (your model) with a name in the form *BayesianModelXXX* (replace XXX by what you want). This class should inherit a base model class either *BayesianModelDefault* or *BayesianModelGeneric* to properly work and be recognized by the extension.
4. Re-implement any method, property you need. In general it will be the `convert_input` one. This method receive as a parameter a *DataToExport* object containing all the data acquired by all selected detectors and should return a float: the target value to be optimized. For more details on the methods to be implemented, see *The Bayesian Extension and utilities*.

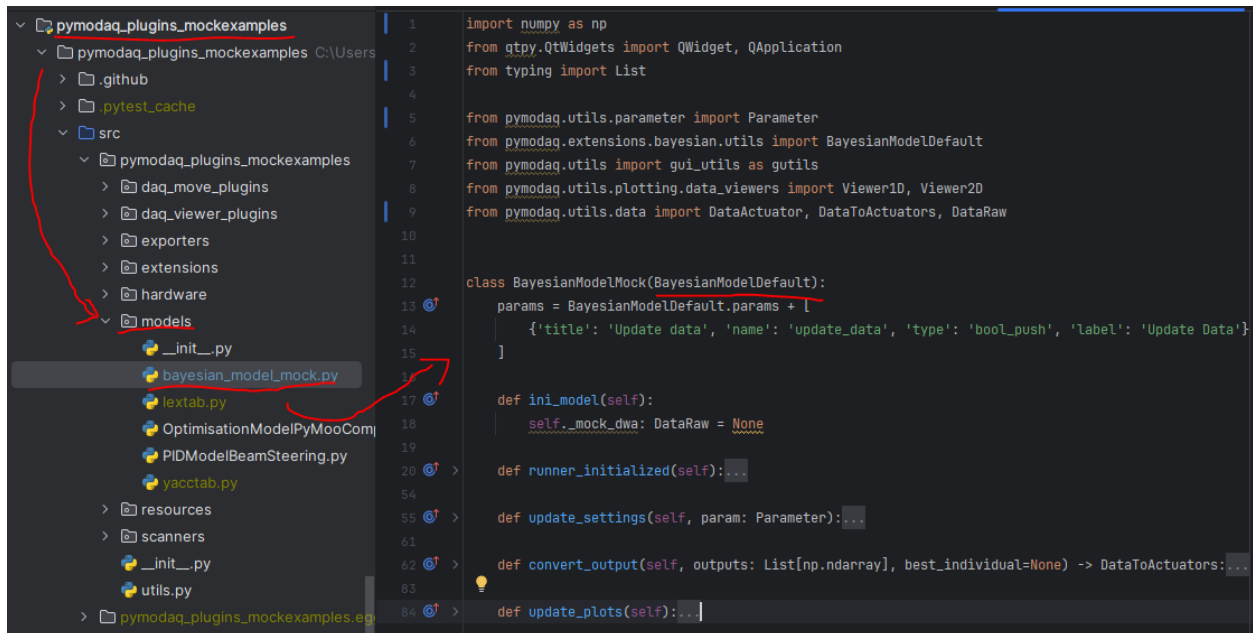


Fig. 7.65: Example of a custom Bayesian model.

H5Browser

Exploring data

The h5 browser is an object that helps browsing of data and metadata. It asks you to select a h5 file and then display a window such as Fig. 7.66. Depending the element of the file you are selecting in the h5 file tree, various metadata can be displayed, such as *scan settings* or *module settings* at the time of saving. When double clicking on data type entries in the tree, the data viewer (type *Plotting all other data* that can display data dimensionality up to 4) will display the selected data node .

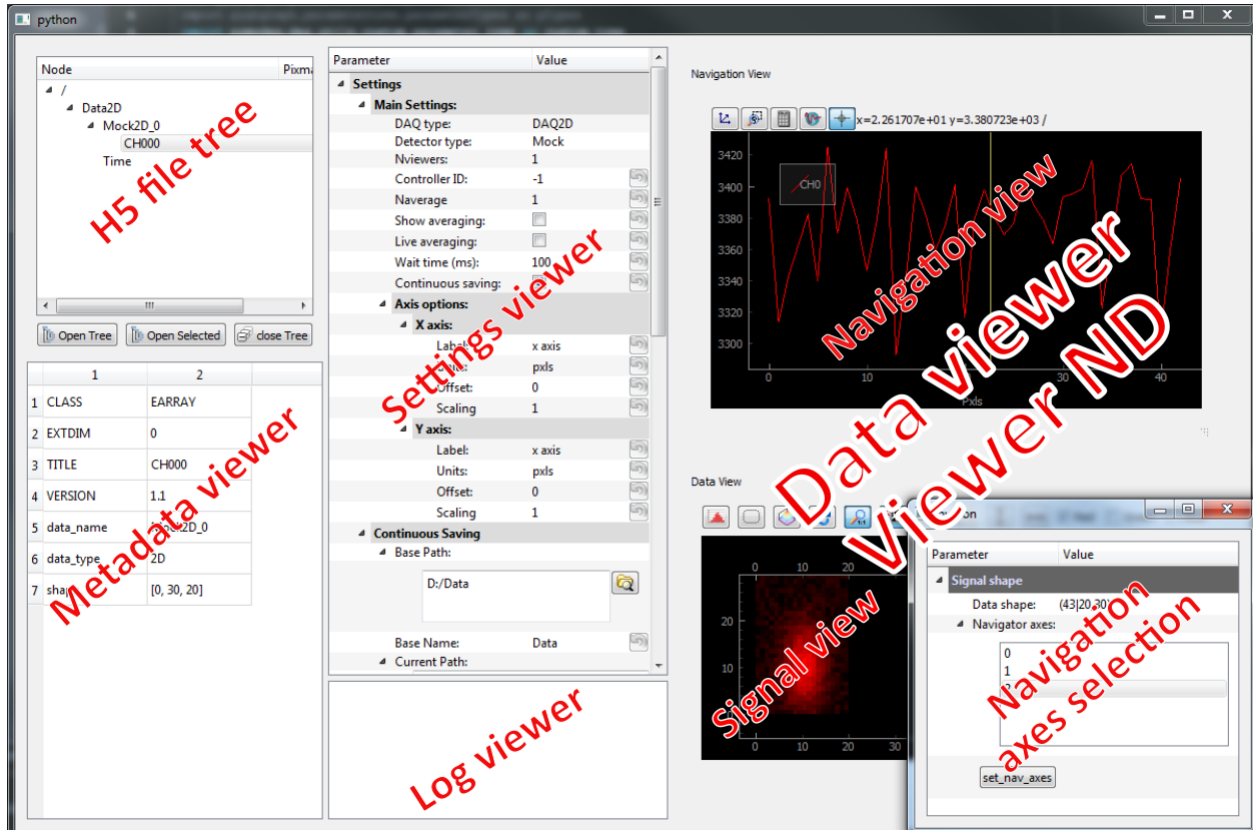


Fig. 7.66: h5 browser to explore saved datas

Some options are available when right clicking on a node, see Fig. 7.67.

- Export as: allow exporting of the data in the selected node to another known file format
- Add Comment: add a comment into the metadata of the node
- Plot Node: plot data (equivalent as double clicking)
- Plot Nodes: plot data hanging from the same channel
- Plot Node with Bkg: plot data with subtracted background (if present)
- Plot Nodes with Bkg: plot data hanging from the same channel with subtracted background (if present)

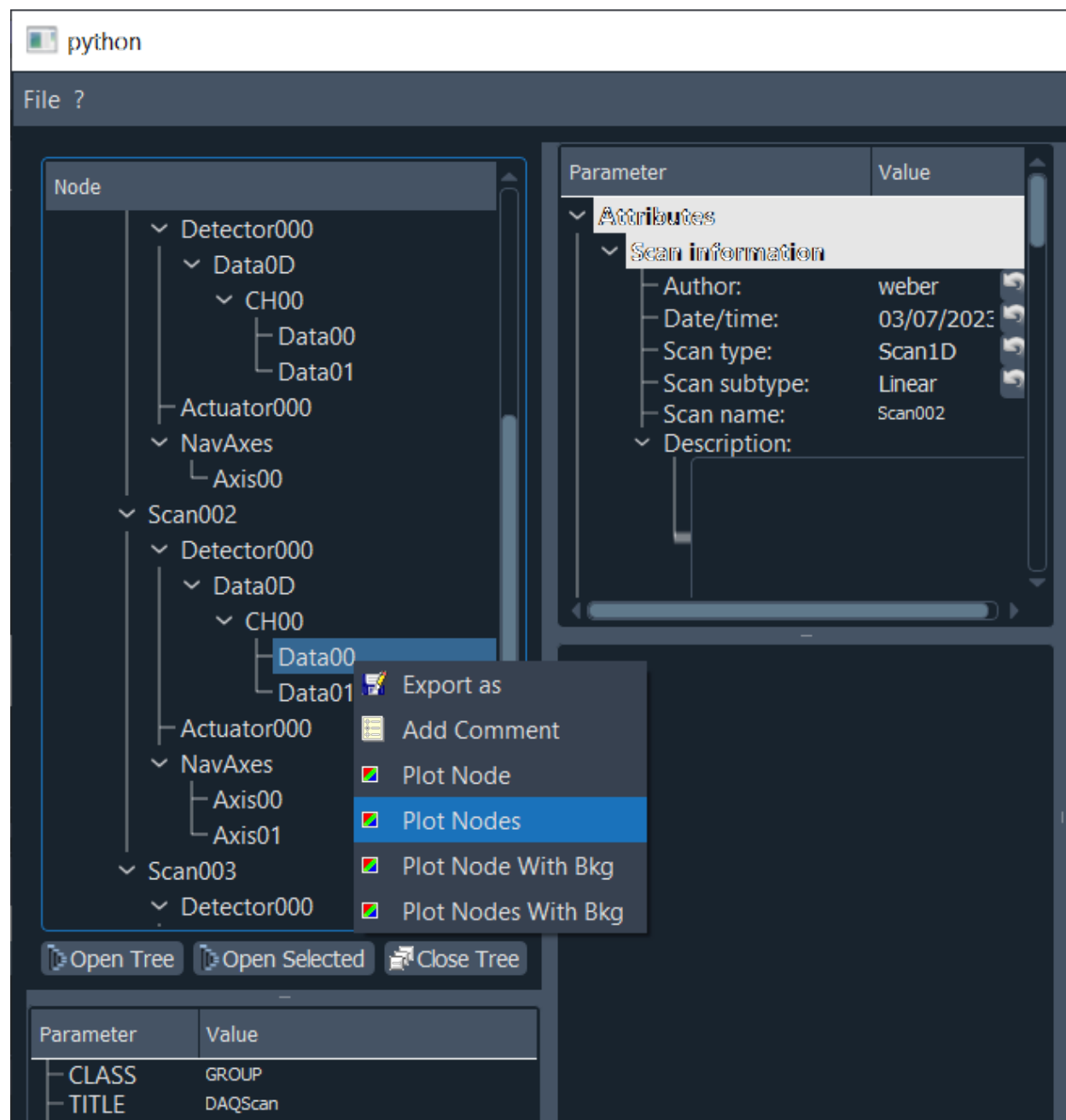


Fig. 7.67: h5 browser options

Associating H5Browser with .h5 files

By default, the H5Browser always asks the user to select a file. One can instead open a specified .h5 file directly, using the `--input` optional command line argument as follows:

```
h5browser --input my_h5_file.h5.
```

One can also associate H5Browser to all .h5 file so that it directly opens a file when double clicking on it. Here is how to do it on Windows. Let us assume that you have a conda environment named *my_env*, in which PyMoDAQ is installed.

In Windows, the path to your conda executable will be something like:

```
C:\Miniconda3\condabin\conda.bat
```

Now that you have written down this path, open your favorite text editing tool (e.g. notepad) and create a file called *H5Opener.bat* (for instance) with the following contents:

```
@ECHO OFF
call C:\Miniconda3\condabin\conda.bat activate my_env
h5browser --input %1
```

Note: The precise path of your environment may be different from the one we wrote just above. Check your conda installation to verify this: *conda info* and *conda env list*

After creating the file, simply right click on any .h5 file, choose **Open with**, *Try an app on this PC*, you should see a list of programs, at the bottom you have to tick *Always use this app to open .h5 files* and then click *Look for another app on this PC*. You can browse to the location of *H5Opener.bat* and you are done. Double clicking any .h5 file will now open the H5Browser directly loading the selected file.

Console

Under construction

7.3.6 Data Management

Data are at the center of the PyMoDAQ ecosystem. From their acquisition up to their saving and plotting, you'll be confronted with them. It is therefore of paramount importance that data objects be well understood and be used transparently by all of PyMoDAQ's modules.

What is PyMoDAQ's Data?

Data in PyMoDAQ are objects with many characteristics:

- a type: float, int, ...
- a dimensionality: Data0D, Data1D, Data2D and we will discuss about *DataND*
- units
- axes
- actual data as numpy arrays
- uncertainty/error bars

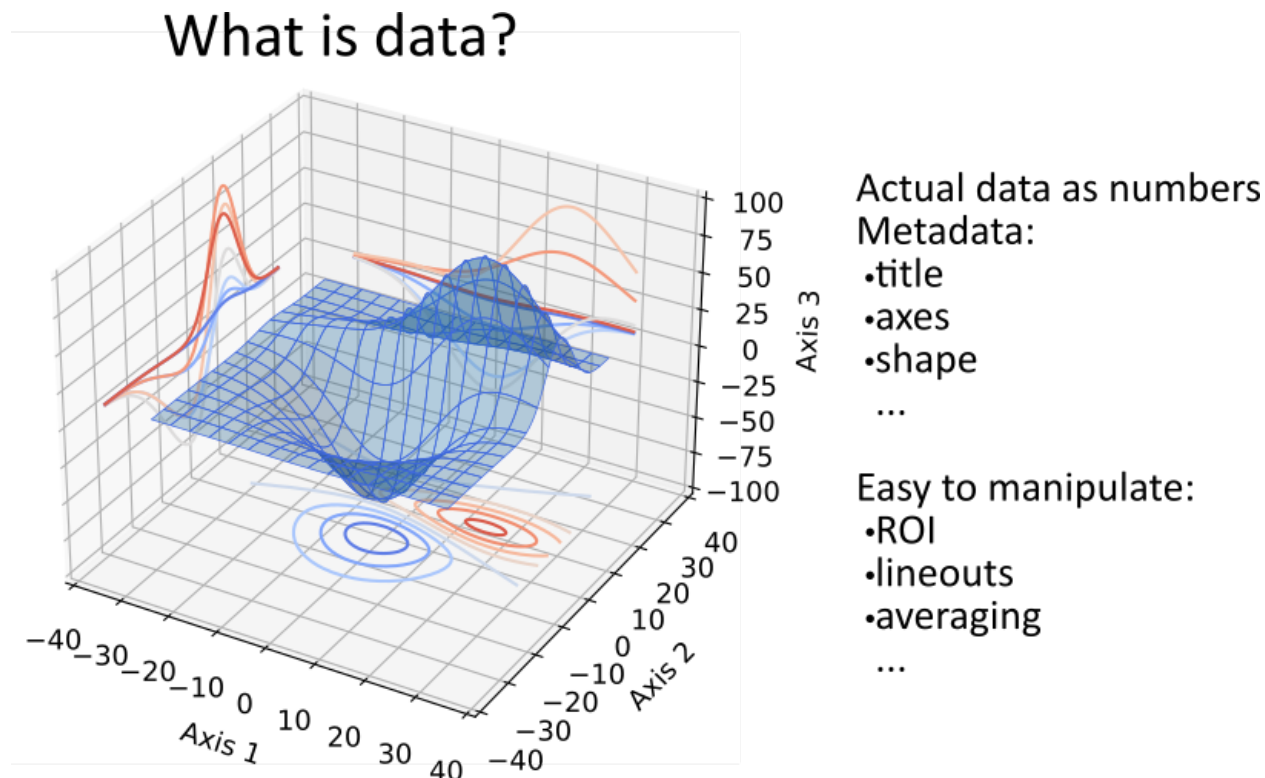


Fig. 7.68: What is PyMoDAQ's data?.

Because of this variety, PyMoDAQ introduce a set of objects including metadata (for instance the time of acquisition) and various methods and properties to manipulate those (getting name, slicing, concatenating...). The most basic object is `DataLowLevel` whose all data objects will inherit. It is very basic and will only store a name as a string and a timestamp from its time of creation.

Then one have `DataBase` objects that stores homogeneous data (data of same type) having the same shape as a list of numpy arrays.

Numpy is fundamental in python and it was obvious to choose that. However, instruments can acquire data having the same type and shape but from different channels. It then makes sense to have a list of numpy arrays.

Figure Fig. 7.69 presents the different types of data objects introduced by PyMoDAQ, which are also described below with examples on how to use them.

DataBase

DataBase, see [Data Management](#), is the most basic object to store data (it should in fact not be used for real cases, please use `DataWithAxes`). It takes as argument a name, a *DataSource*, a *DataDim*, a *DataDistribution*, the actual data as a list of numpy arrays (even for scalars), labels (a name for each element in the list), eventually an origin (a string from which module it originates) and optional named arguments.

```
>>> import numpy as np
>>> from pymodaq.utils.data import DataBase, DataSource, DataDim, DataDistribution
>>> data = DataBase('mydata', source=DataSource['raw'],\
```

(continues on next page)

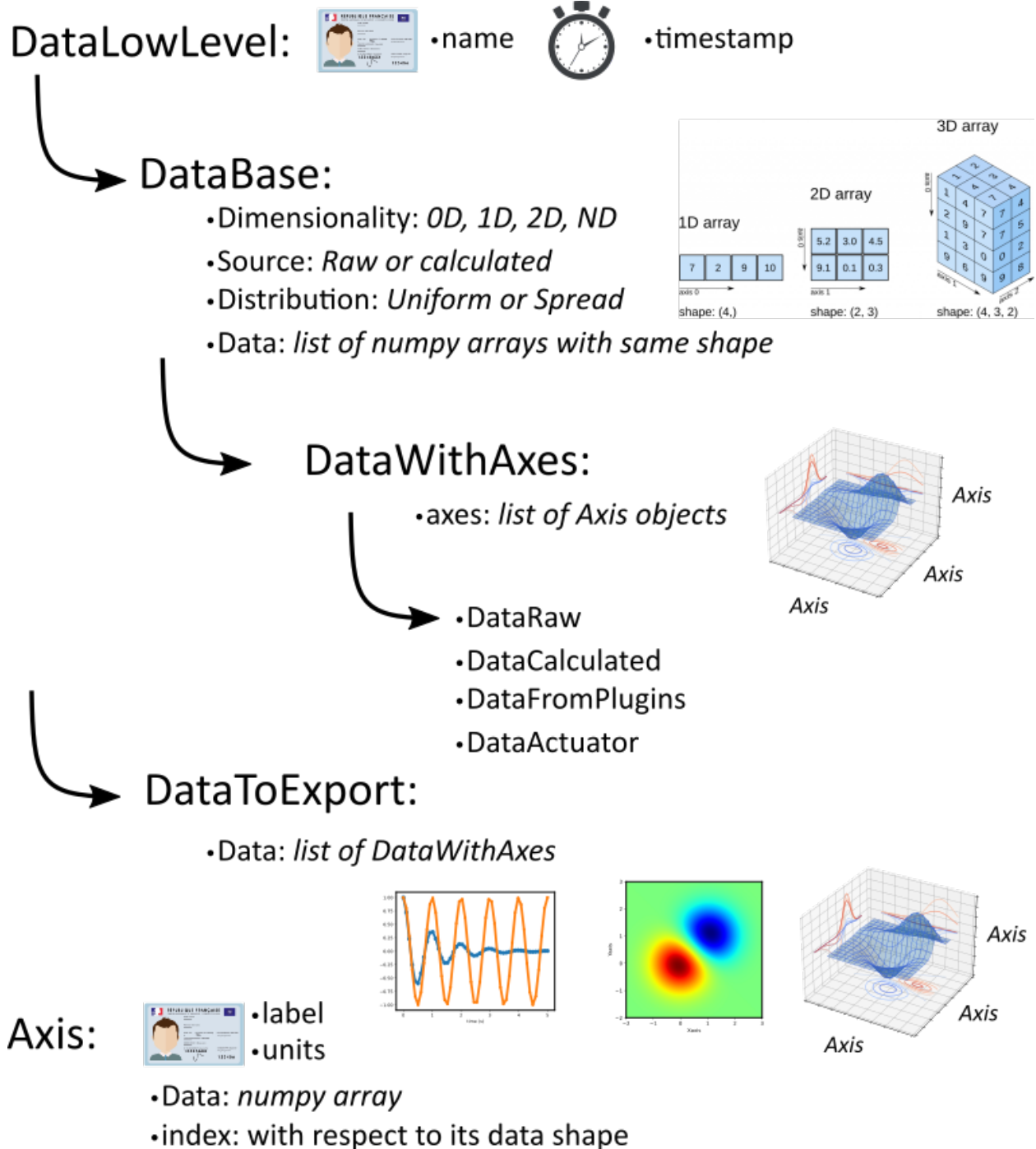


Fig. 7.69: Zoology of PyMoDAQ's data objects.

(continued from previous page)

```
distribution=DataDistribution['uniform'], data=[np.array([1,2,3]), np.array([4,5,6])],\
labels=['channel1', 'channel2'], origin="documentation's code")
```

When instantiated, some checks are performed:

- checking the homogeneity of the data
- the consistency of the dimensionality and the shape of the numpy arrays
- if no dimensionality is given, it is inferred from the data's shape

Useful properties can then be used to check and manipulate the data. For instance one can check the *length* of the object (number of numpy arrays in the list), the *size* (number of elements in the numpy arrays), the *shape* (shape of the numpy arrays).

```
>>> data.dim
<DataDim.Data1D: 1>
>>> data.source
<DataSource.raw: 0>
>>> data.shape
(3,)
>>> data.length
2
>>> data.size
3
```

One can also make mathematical operations between data objects (sum, subtraction, averaging) or appending numpy arrays (of same type and shape) to the data object and iterating over the numpy arrays with the standard *for* loop.

```
>>> for subdata in data:
    print(subdata)
    print(subdata.shape)
[1 2 3]
(3,)
[4 5 6]
(3,)
```

For a full description see *What is PyMoDAQ's Data?*.

Of course for data that are not scalar, a very important information is the axis associated with the data (one axis for waveforms, two for 2D data or more for hyperspectral data). PyMoDAQ therefore introduces *Axis* and *DataWithAxes* objects.

Axis

The *Axis* object stores the information about the data's axis

```
>>> from pymodaq.utils.data import Axis
>>> axis = Axis('myaxis', units='seconds', data=np.array([3,7,11,15]), index=0)
>>> axis
Axis: <label: myaxis> - <units: seconds> - <index: 0>
```

It has a name, units, actual data as a numpy array and an index referring to which dimension of Data the axis is referring to. For example, `index=0` for the vertical axis of 2D data and `index=1` for the horizontal (or inversely, it's up to you...).

Because there is no need to store a linearly spaced array, when instantiated, the `Axis` object will, for linear axis's data replace it by `None` but compute an offset and a scaling factor

```
>>> axis.data
None
>>> axis.offset
3
>>> axis.scaling
4.0
>>> axis.size
4
```

`Axis` object has also properties and methods to manipulate the object, for instance to retrieve the associated numpy array:

```
>>> axis.get_data()
array([ 3.,  7., 11., 15.])
```

and mathematical methods:

```
>>> axis.mean()
11.0
>>> axis.find_index(11.0)
2
```

and a special slicer property to get subparts of the axis's data (but as a new `Axis` object):

```
>>> axis[iaxis[2:]].get_data()
array([11., 15.])
```

DataWithAxes

When dealing with data having axes (even 0D data can be defined as `DataWithAxes`), the `DataBase` object is no more enough to describe the data. PyMoDAQ therefore introduces `DataWithAxes` which inherits from `DataBase` and introduces more metadata and functionalities.

```
>>> from pymodaq.utils.data import DataWithAxes
>>> data = DataWithAxes('mydata', source=DataSource['raw'], dim=DataDim['Data2D'], \
distribution=DataDistribution['uniform'], data=np.array([[1,2,3], [4,5,6]]), \
axes=[Axis('vaxis', index=0, data=np.array([-1, 1])),
Axis('haxis', index=1, data=np.array([10, 11, 12]))])
>>> data
<DataWithAxes, mydata, (|2, 3)>
>>> data.axes
[Axis: <label: vaxis> - <units: > - <index: 0>,
Axis: <label: haxis> - <units: > - <index: 1>]
```

This object has a few more methods and properties related to the presence of axes. It has in particular an `AxesManager` attribute that deals with the `Axis` objects and the Data's representation (`|2, 3`) Here meaning the data has a *signal* shape of `(2, 3)`. The notion of signal will be highlighted in the next paragraph.

It also has a slicer property to get subdata:

```
>>> sub_data = data.isig[1:, 1:]
>>> sub_data.data[0]
array([5, 6])
>>> sub_data = data.isig[:, 1:]
>>> sub_data.data[0]
array([[2, 3],
       [5, 6]])
```

Uncertainty/error bars

The result of a measurement can be captured through averaging of several identical data. This batch of data can be saved as a higher dimensionality data (see [DAQ Scan](#) averaging). However the data could also be represented by the mean of this average and the standard deviation from the mean. *DataWithAxes* introduces therefore this concept as another object attribute: *errors*.

```
data = DataWithAxes('mydata', source=DataSource['raw'], dim=DataDim['Data1D'],
                    data=np.array([1,2,3])),
                    axes=[Axis('axis', index=0, data=np.array([-1, 0, 1])),
                          errors=[np.array([0.01, 0.03, 0,1])])
```

The *errors* parameter should be either None (default) or a list of numpy arrays (list as long as there are data numpy arrays) having the same shape as the actual data.

DataWithAxes and signal/navigation axes

Signal and Navigation is a term taken from the hyperspy package vocabulary. It is useful when dealing with multidimensional data. Imagine data you obtained from a camera (256x1024 pixels) during a linear 1D scan of one actuator (100 steps). The final shape of the data would be (100, 256, 1024). The first dimension corresponds to a Navigation axis (the scan), and the rest to Signal axes (the real detector's data). The corresponding data has a dimensionality of DataND and a representation of (100|256,1024).

This is why *DataWithAxes* can be instantiated with another parameter: *nav_indexes*. This is a tuple containing the index of the axes that should be considered as Navigation. For instance:

```
>>> data = DataWithAxes('mydata', source=DataSource['raw'], dim=DataDim['Data2D'], \
distribution=DataDistribution['uniform'], data=np.array([[1,2,3], [4,5,6]]), \
axes=[Axis('vaxis', index=0, data=np.array([-1, 1])),
      Axis('haxis', index=1, data=np.array([10, 11, 12]))],
nav_indexes = (1,))
```

here because I specified *nav_indexes* as a non-empty tuple, the dimensionality of the data is actually DataND:

```
>>> data.dim
<DataDim.DataND: 3>
```

and the representation shows the navigation/signal parts of the data

```
>>> data
<DataWithAxes, mydata, (3|2)>
```

That is completely controlled from the *nav_indexes* attribute and the corresponding Axis's attribute: *index*.

```
>>> data.nav_indexes = (0,)
>>> data
<DataWithAxes, mydata, (2|3)>
>>> data.sig_indexes
(1,)
```

```
>>> data.nav_indexes = (0, 1)
>>> data
<DataWithAxes, mydata, (2,3|)>
>>> data.sig_indexes
()
```

```
>>> data.nav_indexes = ()
>>> data
<DataWithAxes, mydata, (|2, 3)>
>>> data.dim
<DataDim.Data2D: 2>
>>> data.sig_indexes
(0, 1)
```

When using DataND another slicer property can be used:

```
>>> data.nav_indexes = (0, 1)
>>> sub_data = data.inav[1:, 1:]
>>> sub_data
<DataWithAxes, mydata, (2|)>
>>> sub_data.data[0]
array([5, 6])
```

but `sub_data` is a `DataWithAxes` so could be further sliced also along the signal dimension:

```
>>> data.nav_indexes = (0,)
>>> data
<DataWithAxes, mydata, (2|3)>
>>> data.inav[0]
<DataWithAxes, mydata, (|3)>
>>> data.inav[0].isig[2]
<DataWithAxes, mydata, (|1)>
```

Uniform and Spread Data

So far, everything we've said can be well understood for data taken on a uniform grid (1D, 2D or more). But some scanning possibilities of the `DAQ_Scan` (Tabular) allows to scan on specifics (and possibly random) values of the actuators. In that case the distribution is `DataDistribution['spread']`. Such distribution will be differently plotted and differently saved in a h5file. It's dimensionality will be `DataND` and a specific `AxesManager` will be used. Let's consider an example:

One can take images data (20x30 pixels) as a function of 2 parameters, say xaxis and yaxis non-uniformly spaced

```
>>> data.shape = (150, 20, 30)
>>> data.nav_indexes = (0,)
```

The first dimension (150) corresponds to the navigation (there are 150 non uniform data points taken) The second and third correspond to signal data, here an image of size (20x30 pixels) so:

- `nav_indexes` is (0,)
- `sig_indexes` is (1, 2)

```
>>> xaxis = Axis(name=xaxis, index=0, data=...)
>>> yaxis = Axis(name=yaxis, index=0, data=...)
```

both of length 150 and both referring to the first index (0) of the shape

In fact from such a data shape the number of navigation axes is unknown . In our example, they are 2. To somehow keep track of some ordering in these navigation axes, one adds an attribute to the `Axis` object: the `spread_order`

```
>>> xaxis = Axis(name=xaxis, index=0, spread_order=0, data=...)
>>> yaxis = Axis(name=yaxis, index=0, spread_order=1, data=...)
```

This ordering will be very important for plotting of the data, see for instance below for an adaptive scan:

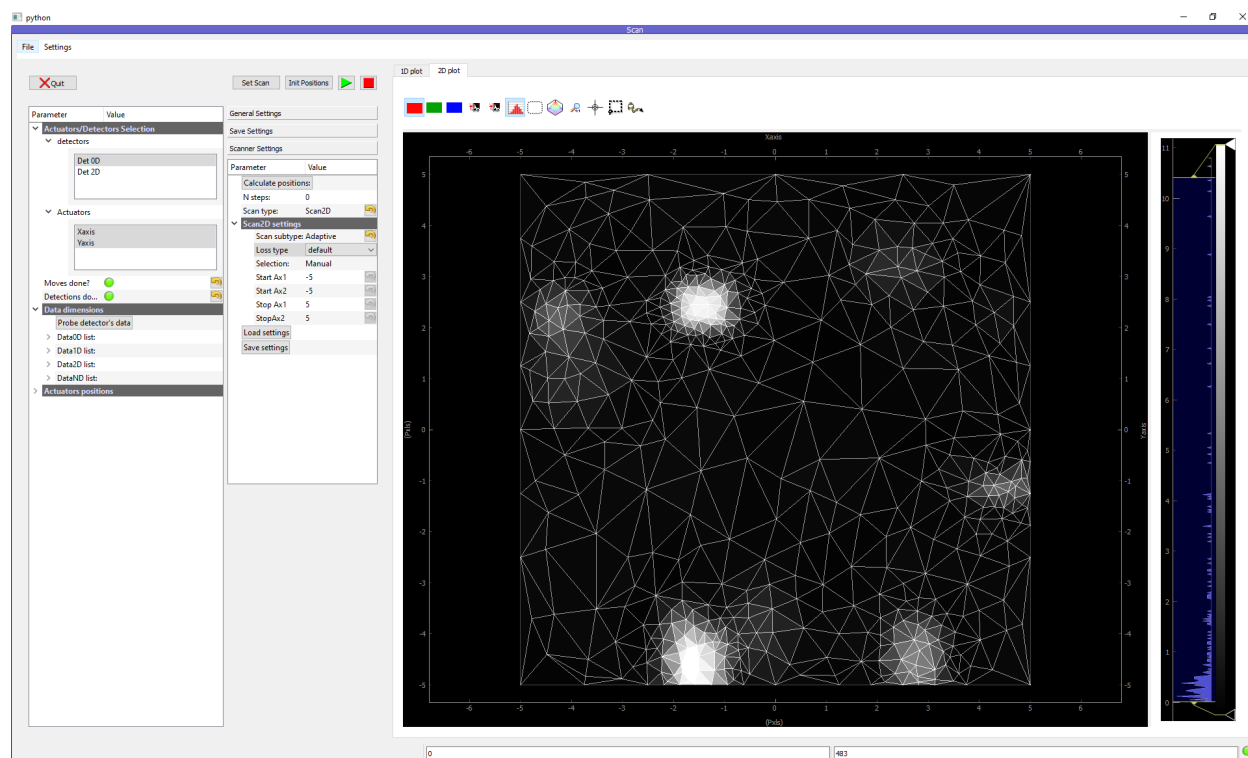


Fig. 7.70: Non uniform 2D plotting of Spread DataWithAxes.

Special DataWithAxes

For explicit meaning, several classes are inheriting `DataWithAxes` with adhoc attributes such as:

- `DataRaw`: `DataWithAxes` with its source set to `DataSource['raw']`
- `DataFromPlugins`: explicit `DataRaw` to be used within Instrument plugins
- `DataCalculated`: `DataWithAxes` with its source set to `DataSource['calculated']`
- `DataFromRoi`: explicit `DataCalculated` to be used when processing data using ROI.

DataToExport

In general a given instrument (hence its PyMoDAQ's Instrument plugin) will generate similar data (for instance several `Data1D` waveforms for each channel of an oscilloscope). Such data can be completely defined using `DataWithAxes` as we saw above.

However, when then plotting such data, the user can decide to use ROI to extract some meaningful information to be displayed in a live DAQ_Scan plot. This means that the corresponding DAQ_Viewer will produce both `Data1D`'s data but also several `Data0D`'s ones depending on the number of used ROIs. To export (emit signals) or save (to h5), it would be much better to have a specialized object to deal with these non-similar data. This is the role of the `DataToExport` object.

`DataToExport` is a `DataLowLevel` object with an extra attribute `data`, that is actually a list of `DataWithAxes` objects:

```
>>> from pymodaq.utils.data import DataToExport, DataRaw
>>> dwa0D = DataRaw('dwa0D', data=[np.array([1]), np.array([2]) , np.array([3])])
>>> dwa1D = DataRaw('dwa1D', data=[np.array([1, 2 , 3])])
>>> dte = DataToExport(name='a_lot_of_different_data', data=[dwa0D, dwa1D])
>>> dte
DataToExport: a_lot_of_different_data <len:2>
```

It has a length of 2 because contains 2 `DataWithAxes` objects (`dwa`). One can then easily get the data from it :

```
>>> dte[0]
<DataRaw, dwa0D, (|1)>
```

or get `dwa` from their dimensionality, their name, the number of axes they have ...

```
>>> dte.get_data_from_dim('Data1D').data[0]
<DataRaw, dwa1D, (|3)>
>>> dte.get_names()
['dwa0D', 'dwa1D']
>>> dte.get_data_from_name('dwa0D')
<DataRaw, dwa0D, (|1)>
```

`Dwa` can also be appended or removed to/from a `DataToExport`.

For more details see [Union of Data](#)

Saving and loading data

Datas saved using PyMoDAQ, either the *DAQ_Scan* or the *DAQ_Viewer* modules or others, use a binary format known as *hdf5*. This format was originally developed to save big volume of datas from large instruments. Its structure is hierarchical (a bit as folder trees) and one can add metadata to all entries in the tree. For instance, the data type, shape but also some more complex info such as all the settings related to a module or an instrument plugin. This gives a unique file containing both data and metadata.

Python wrappers around the HDF5 library (hdf5 backends) are available, such as *h5py* or *pytables* (default one used by PyMoDAQ). For an even easier use, PyMoDAQ also has a dedicated object allowing a transparent use of any hdf5 backend: *Hdf5 backends*. It also has an object used for saving data: *Low Level saving* and browsing data: *H5Browser*.

These low level objects allow to interact with PyMoDAQ's data and hdf5 file but because displaying and loading correctly data need a specific layout and metadata in the hdf5 file, higher level objects should be systematically used to save and load data. They insure that any data loaded from the hdf5 file will have a correct type: *DataWithAxes* or *DataToExport* and that these data objects will be saved with the appropriate layout and metadata to insure their reconstruction when loading. These objects are defined in the `pymodaq.utils.h5modules.data_saving` module. Their specificity is described below but for a more detailed description, see *High Level saving/loading*.

All these high level saving objects have under the hood a *H5Saver* object dealing with the actual saving. User interface related to saving in PyMoDAQ all use the *H5Saver* *ParameterTree* and settings associated with to control what/where/how to save data, see *H5Saver*.

DataSaver/DataLoader

Saving and loading data objects is a symmetrical action, therefore PyMoDAQ defines objects to do both. These objects all derive from a base class allowing the manipulation of the node (*DataManagement* object), then the child class should define a *data type* and will be responsible for saving and loading such data. Data type means here one of the three main type of PyMoDAQ's data system: *Axis*, *DataWithAxes* or *DataToExport*. These child objects are respectively: *AxisSaverLoader*, *DataSaverLoader* and *DataToExportSaver*.

They all take as initial parameter a *h5saver* object (used to initialize a hdf5 file, see *Low Level saving*), then define specific methods to save their data type. Examples:

AxisSaverLoader

First I create a hdf5 file using the *H5Saver* (here *H5SaverLowLevel* because I'm not in a Qt event loop)

```
>>> import numpy as np
>>> from pathlib import Path
>>> from pymodaq.utils.data import Axis
>>> from pymodaq.utils.h5modules.saving import H5SaverLowLevel
>>> h5saver = H5SaverLowLevel()
>>> h5saver.init_file(Path('atemporaryfile.h5'))
```

Then I create the *Axis* object and its saver/loader

```
>>> from pymodaq.utils.h5modules.data_saving import AxisSaverLoader
>>> axis = Axis('myaxis', units='seconds', data=np.array([3,7,11,15]), index=0)
>>> axis_saver = AxisSaverLoader(h5saver)
```

I save the *Axis* object in the */RawData* node (always created using *H5Saver*)

```
>>> axis_saver.add_axis('/RawData', axis)
/RawData/Axis00 (CARRAY) 'myaxis'
      shape := (4,)
      dtype := float64
```

I can check the content of the file:

```
>>> for node in h5saver.walk_nodes('/'):
>>>     print(node)
/ (GROUP) 'PyMoDAQ file'
/RawData (GROUP) 'Data from PyMoDAQ modules'
/RawData/Logger (VLARRAY) ''
/RawData/Axis00 (CARRAY) 'myaxis'
```

And load back from it, an Axis object identical to the initial one (but not the same one)

```
>>> loaded_axis = axis_saver.load_axis('/RawData/Axis00')
>>> loaded_axis
Axis: <label: myaxis> - <units: seconds> - <index: 0>
>>> loaded_axis == axis
True
>>> loaded_axis is axis
False
```

DataSaverLoader

The DataSaverLoader object will behave similarly with DataWithAxes objects, introducing the methods:

- add_data
- load_data

with a slight asymmetry between the two if one want to load background subtracted data previously saved using the specialized BkgSaver. This guy is identical to the DataSaverLoader except it considers the DataWithAxes to be saved as background data type.

Here I create my data and background object:

```
>>> from pymodaq.utils.data import DataWithAxes, DataSource, DataDim, DataDistribution
>>> data = DataWithAxes('mydata', source=DataSource['raw'], dim=DataDim['Data2D'], \
distribution=DataDistribution['uniform'], data=np.array([[1,2,3], [4,5,6]]), \
axes=[Axis('vaxis', index=0, data=np.array([-1, 1])),
Axis('haxis', index=1, data=np.array([10, 11, 12]))])
>>> bkg = data.deepcopy()
>>> data
<DataWithAxes, mydata, (|2, 3)>
>>> bkg
<DataWithAxes, mydata, (|2, 3)>
```

I add a *detector* node in the h5file:

```
>>> h5saver.add_det_group('/RawData', 'Example')
/RawData/Detector000 (GROUP) 'Example'
  children := []
```

and save in this node the data:

```
>>> from pymodaq.utils.h5modules.data_saving import DataSaverLoader
>>> datasaver = DataSaverLoader(h5saver)
>>> datasaver.add_data('/RawData/Detector000', data)
```

and check the file content:

```
>>> for node in h5saver.walk_nodes('/'):
>>>     print(node)
/ (GROUP) 'PyMoDAQ file'
/RawData (GROUP) 'Data from PyMoDAQ modules'
/Axis00 (CARRAY) 'myaxis'
/RawData/Logger (VLARRAY) ''
/RawData/Detector000 (GROUP) 'Example'
/RawData/Detector000/Data00 (CARRAY) 'mydata'
/RawData/Detector000/Axis00 (CARRAY) 'vaxis'
/RawData/Detector000/Axis01 (CARRAY) 'haxis'
```

It saved automatically the Axis objects associated with the data

```
>>> loaded_data = datasaver.load_data('/RawData/Detector000/Data00')
>>> loaded_data
<DataWithAxes, mydata, (|2, 3)>
>>> loaded_data == data
True
>>> loaded_data is data
False
```

Now about the background:

```
>>> from pymodaq.utils.h5modules.data_saving import BkgSaver
>>> bkgsaver = BkgSaver(h5saver)
>>> bkgsaver.add_data('/RawData/Detector000', data, save_axes=False)
```

no need to save the axes as they are shared between data and its background

```
>>> for node in h5saver.walk_nodes('/RawData/Detector000'):
>>>     print(node)
/RawData/Detector000 (GROUP) 'Example'
/RawData/Detector000/Data00 (CARRAY) 'mydata'
/RawData/Detector000/Axis00 (CARRAY) 'vaxis'
/RawData/Detector000/Axis01 (CARRAY) 'haxis'
/RawData/Detector000/Bkg00 (CARRAY) 'mydata'
```

I now have a Bkg data type and can load data with or without bkg included:

```
>>> loaded_data_bkg = datasaver.load_data('/RawData/Detector000/Data00', with_bkg=True)
>>> loaded_data_bkg
<DataWithAxes, mydata, (|2, 3)>
>>> loaded_data_bkg == loaded_data
False
>>> loaded_data_bkg.data[0]
array([[0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0]])
>>> loaded_data.data[0]
array([[1, 2, 3],
       [4, 5, 6]])
```

DataToExportSaver

Finally the same apply for DataToExport containing multiple DataWithAxes. Its associated DataToExportSaver will save its data into different channel nodes themselves filtered by dimension. The only difference here, is that it won't be able to load the data back to a dte

Let's say I create a DataToExport containing 0D, 1D and 2D DataWithAxes (see the tests file):

```
>>> dte = DataToExport(name='mybigdata', data=[data2D, data0D, data1D, data0Dbis])
>>> from pymodaq.utils.h5modules.data_saving import DataToExportSaver
>>> dte_saver = DataToExportSaver(h5saver)
```

```
>>> h5saver.add_det_group('/RawData', 'Example dte')
/RawData/Detector001 (GROUP) 'Example dte'
  children := []
```

```
>>> dte_saver.add_data('/RawData/Detector001', dte)
```

```
>>> for node in h5saver.walk_nodes('/RawData/Detector001'):
>>>     print(node)
/RawData/Detector001 (GROUP) 'Example dte'
/RawData/Detector001/Data0D (GROUP) ''
/RawData/Detector001/Data1D (GROUP) ''
/RawData/Detector001/Data2D (GROUP) ''
/RawData/Detector001/Data0D/CH00 (GROUP) 'mydata0D'
/RawData/Detector001/Data0D/CH01 (GROUP) 'mydata0Dbis'
/RawData/Detector001/Data1D/CH00 (GROUP) 'mydata1D'
/RawData/Detector001/Data2D/CH00 (GROUP) 'mydata2D'
/RawData/Detector001/Data2D/CH00/Data00 (CARRAY) 'mydata2D'
/RawData/Detector001/Data2D/CH00/Data01 (CARRAY) 'mydata2D'
/RawData/Detector001/Data2D/CH00/Axis00 (CARRAY) 'myaxis0'
/RawData/Detector001/Data2D/CH00/Axis01 (CARRAY) 'myaxis1'
/RawData/Detector001/Data1D/CH00/Data00 (CARRAY) 'mydata1D'
/RawData/Detector001/Data1D/CH00/Data01 (CARRAY) 'mydata1D'
/RawData/Detector001/Data1D/CH00/Axis00 (CARRAY) 'myaxis0'
/RawData/Detector001/Data0D/CH00/Data00 (CARRAY) 'mydata0D'
/RawData/Detector001/Data0D/CH00/Data01 (CARRAY) 'mydata0D'
/RawData/Detector001/Data0D/CH01/Data00 (CARRAY) 'mydata0Dbis'
/RawData/Detector001/Data0D/CH01/Data01 (CARRAY) 'mydata0Dbis'
```

Here a bunch of nodes has been created to store all the data present in the dte object.

DataLoader

If one want to load several nodes at ones or include the navigation axes saved at the root of the nodes, one should use the `DataLoader` that has methods to load one `DataWithAxes` (including eventual navigation axes) or a bunch of it into a `DataToExport`:

- `load_data` -> `DataWithAxes`
- `load_all` -> `DataToExport`

Special DataSaver

Some more dedicated objects are derived from the objects above. They allow to add Extended arrays (arrays that will be populated after creation, for instance for a scan) and Enlargeable arrays (whose final length is not known at the moment of creation, for instance when logging or continuously saving) see *Specific data class saver/loader*.

Module Savers

Data saved from the various PyMoDAQ's modules should follow a particular layout. For instance grouped in a *Detector* node for data from the `DAQ_Viewer` modules or a *Scan* node for data from the `DAQ_Scan` module. This node also has metadata such as the settings of the `DAQ_Viewer` at the time when the data have been saved. Special layouts and special saver objects are available for each module able to save data: *DAQ Viewer*, *DAQ Move*, *DAQ Scan* and *DAQ Logger*. See *Module savers* for the related objects.

All of these objects inherit from the `ModuleSaver` base class that implements common methods for all savers. Specific saver, such as the `DetectorSaver` then defines a `GroupType`:

```
class GroupType(BaseEnum):
    detector = 0
    actuator = 1
    data = 2
    ch = 3
    scan = 4
    external_h5 = 5
    data_dim = 6
    data_logger = 7
```

This correspond to a particular type of group node in the h5 file. For what we are discussing the relevant group types are *detector*, *actuator*, *scan* and *data_logger*. For the `DetectorSaver` the group type is therefore: `detector`. Once instanced these objects can be attributed with a given `H5Saver` instance. for instance, when saving snapshots from the `DAQ_Viewer`, this code is called:

```
path = 'a/custom/path/for/a/hdf5/file.h5'

h5saver = H5Saver(save_type='detector')
h5saver.init_file(update_h5=True, custom_naming=False, addhoc_file_path=path)

self.module_and_data_saver = module_saving.DetectorSaver(self)
self.module_and_data_saver.h5saver = h5saver
```

Then `self.module_and_data_saver` will automatically create a dedicated group node in the h5 file. Then it can call specific methods to add properly formatted data in the hdf5 file:

```
detector_node = self.module_and_data_saver.get_set_node(where)
self.module_and_data_saver.add_data(detector_node, data, **kwargs)
```

where `data` is a `DataToExport` object (containing possibly multiple `DataWithAxes` objects). The content of such a file can be displayed using the *H5Browser* as shown on figure Fig. 7.71

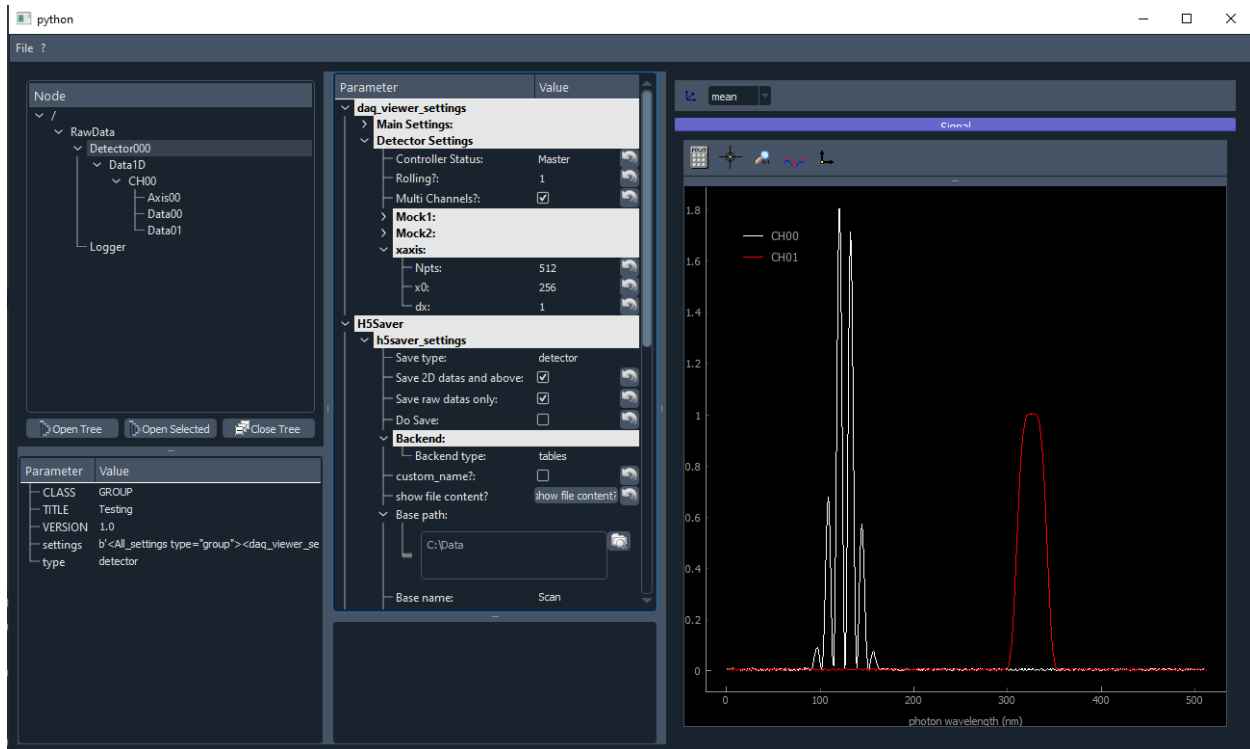


Fig. 7.71: HDF5 file content containing a single `DataWithAxes` (with two channels) saved using the *DetectorSaver* object

One clearly see the layout with the `Detector000` group node (with the setting metadata displayed on the right in a `ParameterTree`), the grouping of data by dimensionality, both channels having the same Axis grouped in the `CH00` group node. Both channels are plotted on the right panel in a `Viewer1D` object.

If multiple `DataWithAxes` were contained in the `DataToExport` they would be stored within `CH00` and `CH01` group nodes as shown in Fig. 7.72 together with their axes and even here with their background

The code used to add the background is:

```
self.module_and_data_saver.add_bkg(detector_node, self._bkg)
```

where `self._bkg` is a `DataToExport` similar to the one we saved but containing background data.

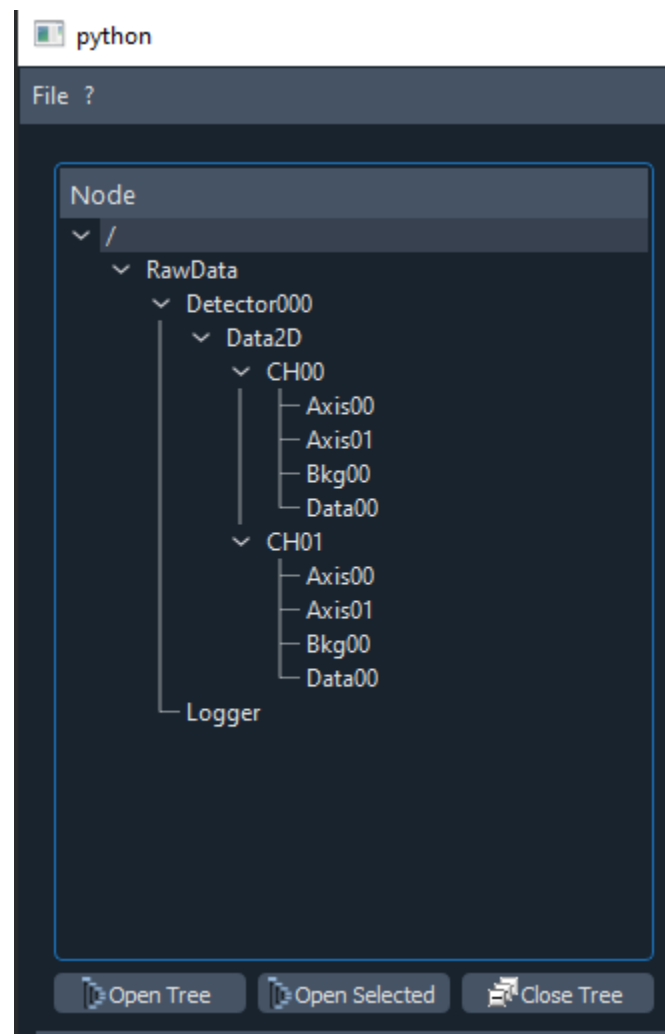


Fig. 7.72: HDF5 file content containing two `DataWithAxes` (with one channel each) saved using the `DetectorSaver` object. They are stored within `CH00` and `CH01` group nodes each with their axes and even here with their background.

Plotting Data

Data in PyMoDAQ are featured with a lot of metadata, allowing their proper description and enabling seamlessly saving/loading to hdf5 files. But what about representation? Analysis? Exploration?

With Python you usually do this by writing a script and manipulate and plot your data using your favorite backend (matplotlib, plotly, qt, tkinter, ...) However because PyMoDAQ is highly graphical you won't need that. PyMoDAQ is featured with various data viewers allowing you to plot any kind of data. You'll see below some nice examples of how to plot your PyMoDAQ's data using the builtin data viewers.

Note: The content of this chapter is available as a [notebook](#).

To execute this notebook properly, you'll need PyMoDAQ >= 4.0.2 (if not released yet, you can get it from [github](#))

Plotting scalars: Viewer0D

Scalars or *Data0D* data are the simplest form of data. However, just displaying numbers is somewhat lacking (in particular when one want to compare data evolution over time, or parameter change...). This is why it is important to keep track of the history of the scalar values. The *Viewer0D*, see below, has such an history as well as tools to keep track of the maximal reached value.

```
%gui qt
import numpy as np

from pymodaq.utils.plotting.data_viewers.viewer0D import Viewer0D
from pymodaq.utils.data import DataRaw

dwa = DataRaw('my_scalar', data=[np.array([10.6]), np.array([-4.6])],
              labels=['scalar1', 'scalar2'])
viewer0D = Viewer0D()
```

```
viewer0D.show_data(dwa)
```

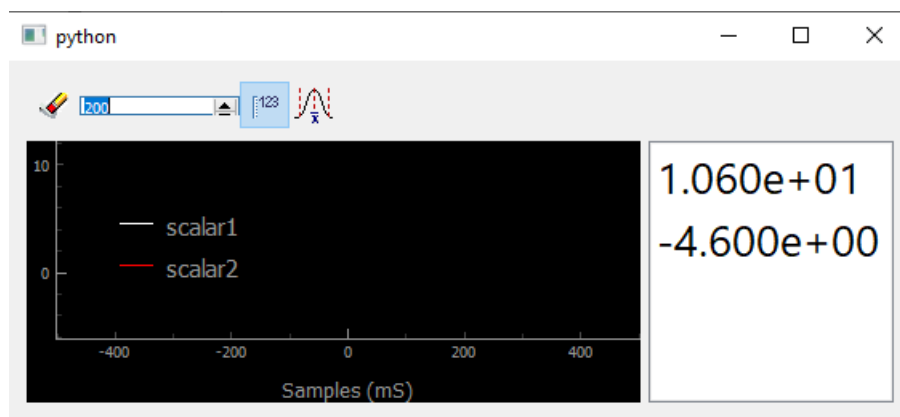



Fig. 7.73: Showing scalars in a *Viewer0D*

Well not much can be seen except for the numbers printed on the right (shown by clicking on the dedicated button ). But what if I call several times the *show_data* method to display evolving signal?

Note: We recall that a *DataRaw* is a particular case of a more generic *DataWithAxes* (*dwa* in short) having its source set to *raw*

```
for ind in range(100):
    dwa = DataRaw('my_scalar', data=[np.sin([ind / 100 * 2*np.pi]),
                                      np.sin([ind / 100 * 2*np.pi + np.pi/4])],
                  labels=['mysinus', 'my_dephased_sinus'])
    viewer0D.show_data(dwa)
```

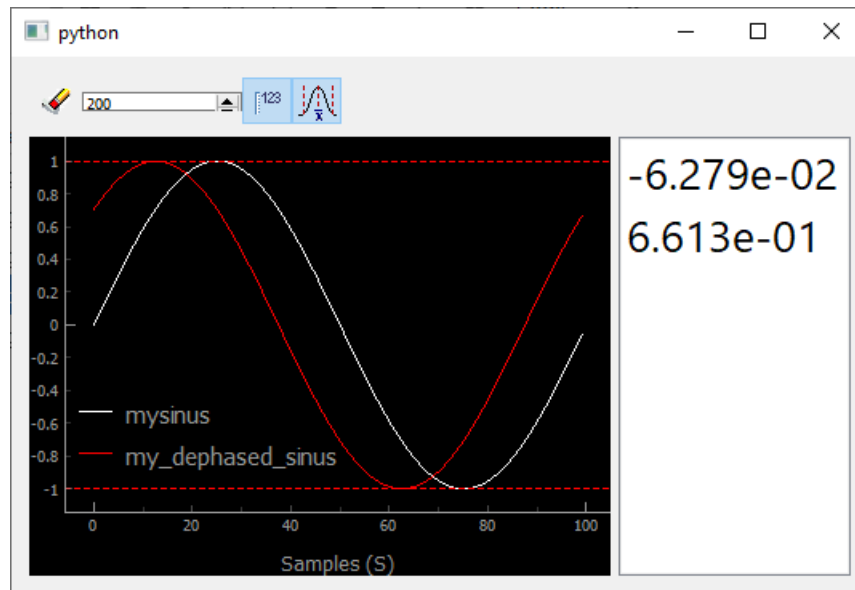



Fig. 7.74: Showing an history of scalars, together with their min and max values (dashed lines)

You immediately see the usefulness of such an history, allowing for instance to optimize signals when tweaking a parameter especially if you use the dashed lines, triggered by , showing the values of the min and max reached values.

Plotting vectors/waveforms: Viewer1D

When increasing complexity, one get one dimensional data. It has one more important metadata, its axis. Properly defining the data object will translate into rich plots:

```
from pymodaq.utils import math_utils as mutils
from pymodaq.utils.data import Axis

axis = Axis('my axis', units='my units', data=np.linspace(-10000, 10000, 100))

dwa1D = DataRaw('my_1D_data', data=[mutils.gauss1D(axis.get_data(), 3300, 2500),
                                    mutils.gauss1D(axis.get_data(), -4000, 1500) * 0.5],
                labels=['a gaussian', 'another gaussian'],
                axes=[axis],
```

(continues on next page)

(continued from previous page)

```
errors=[0.1* np.random.random_sample((axis.size,)) for _ in range(2)]
dwa.plot('qt')
```

Note: One can directly call the method *plot* on a data object, PyMoDAQ will determine which data viewer to use.

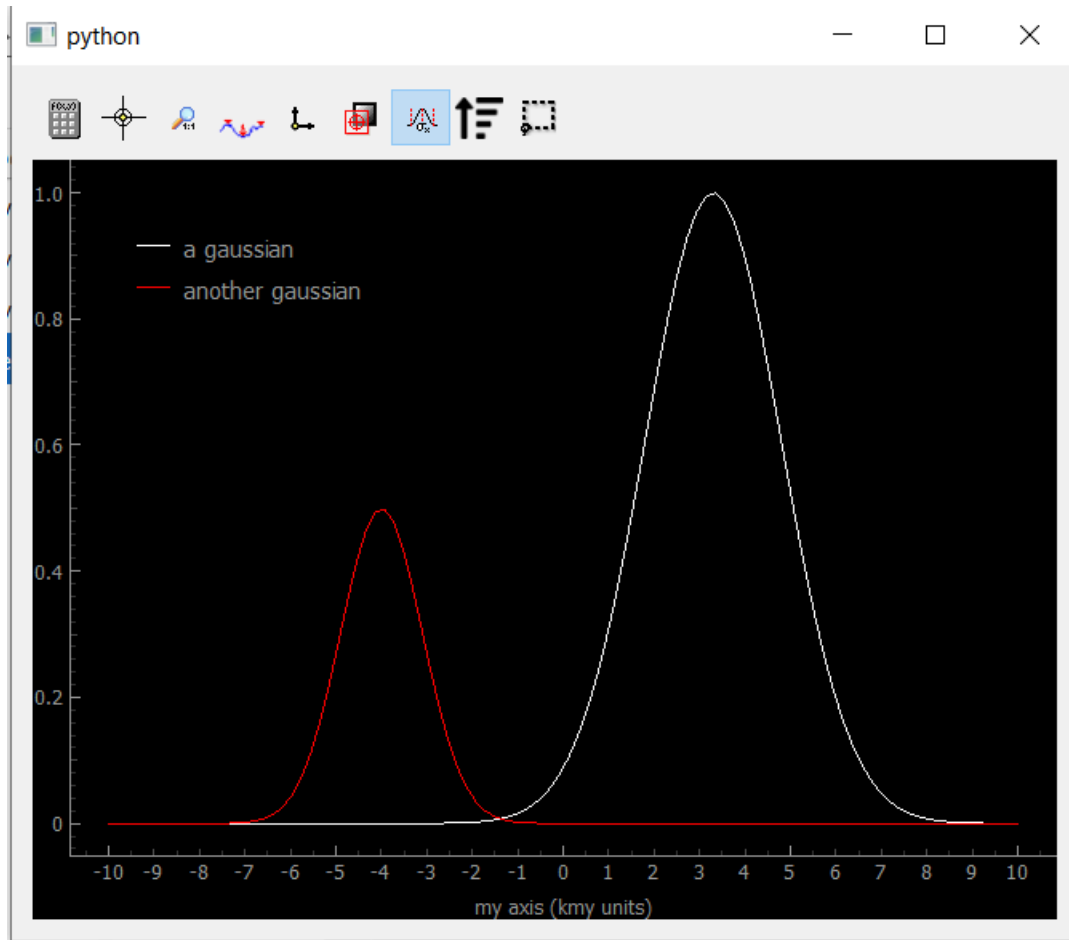











Fig. 7.75: Showing Data1D

You can see the legends correspond to the data labels, while the axis shows both the label and the units in scientific notation (notice the k before 'my units' standing for kilo).

As for the buttons in the toolbar (you can try them from the notebook):

-  : opens the ROI (region of interest) manager, to load, save and define ROI to apply to the data. This will create cropped Data0D from the application of an operation on the cropped data such as *mean*, *sum*, *std*... See figure below, showing the mean value on the bottom panel. ROI can be applied to one of the trace or to both as reflected by the legends
-  : activate the crosshair (yellow vertical line) that can be grabbed and translated. The data at the crosshair position is printed on the right of the toolbar.

- : fix the horizontal/vertical aspect ratio (usefull for xy plot see below)
- : as shown on the figure below, one can switch between solid line or only dots.
- : when data contains two waveforms, using this button will display them in XY mode.
- : when activated, an overlay of the current data will be depicted with a dash line.
- : if the axis data is not monotonous, data will be represented as a scrambled solid line, using this button will reorder the data by ascending values of its axis. See below and figure xx
- : when activated, will display errors (error bars) in the form of a area around the curve
- : extra ROI that can be used independantly of the ROI manager

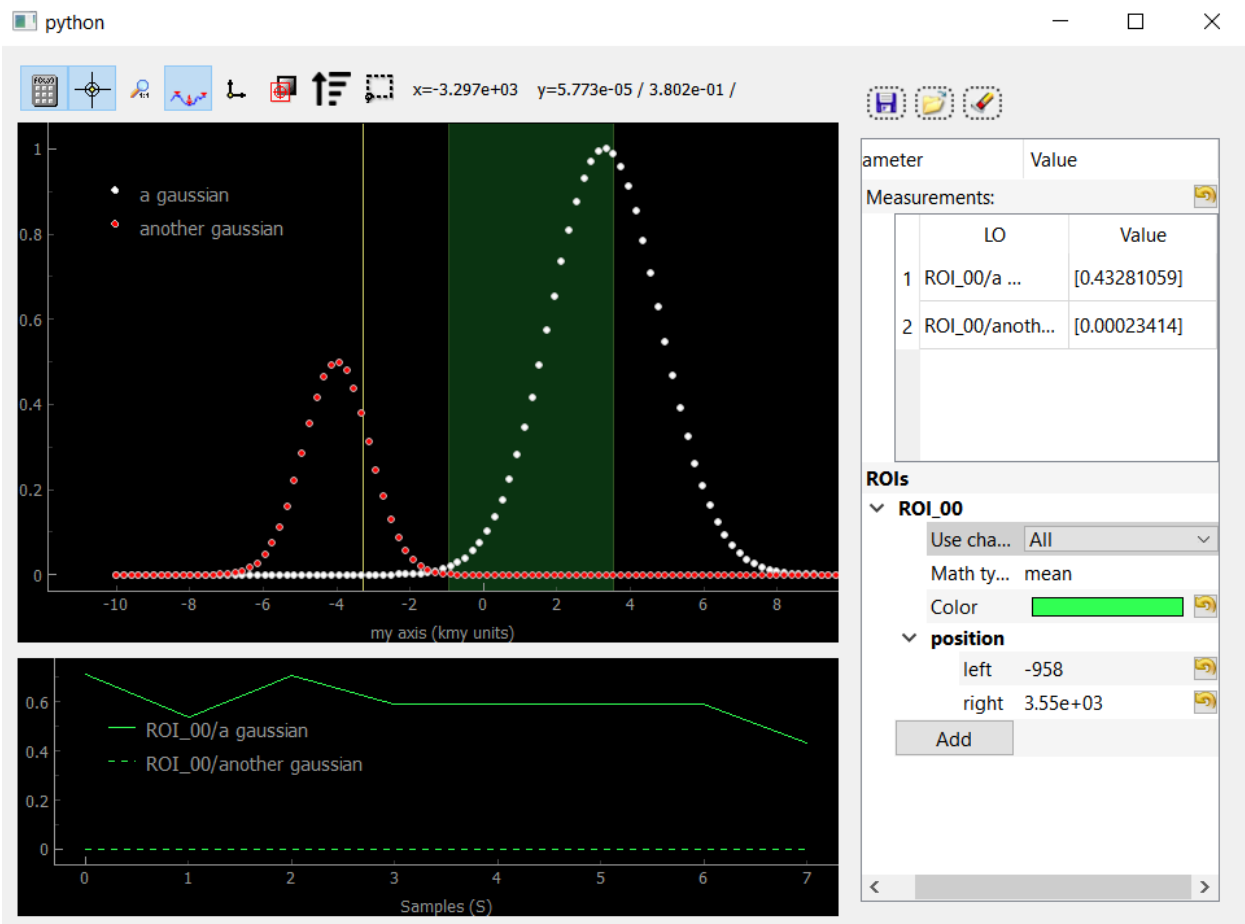


Fig. 7.76: Showing Data1D as dots and with an activated ROI and crosshair

If *Uncertainty/error bars* are defined in the data object, the Viewer1D can easily plot them:

If the axis data is not monotonous, data will be represented as a scrambled solid line, for instance:

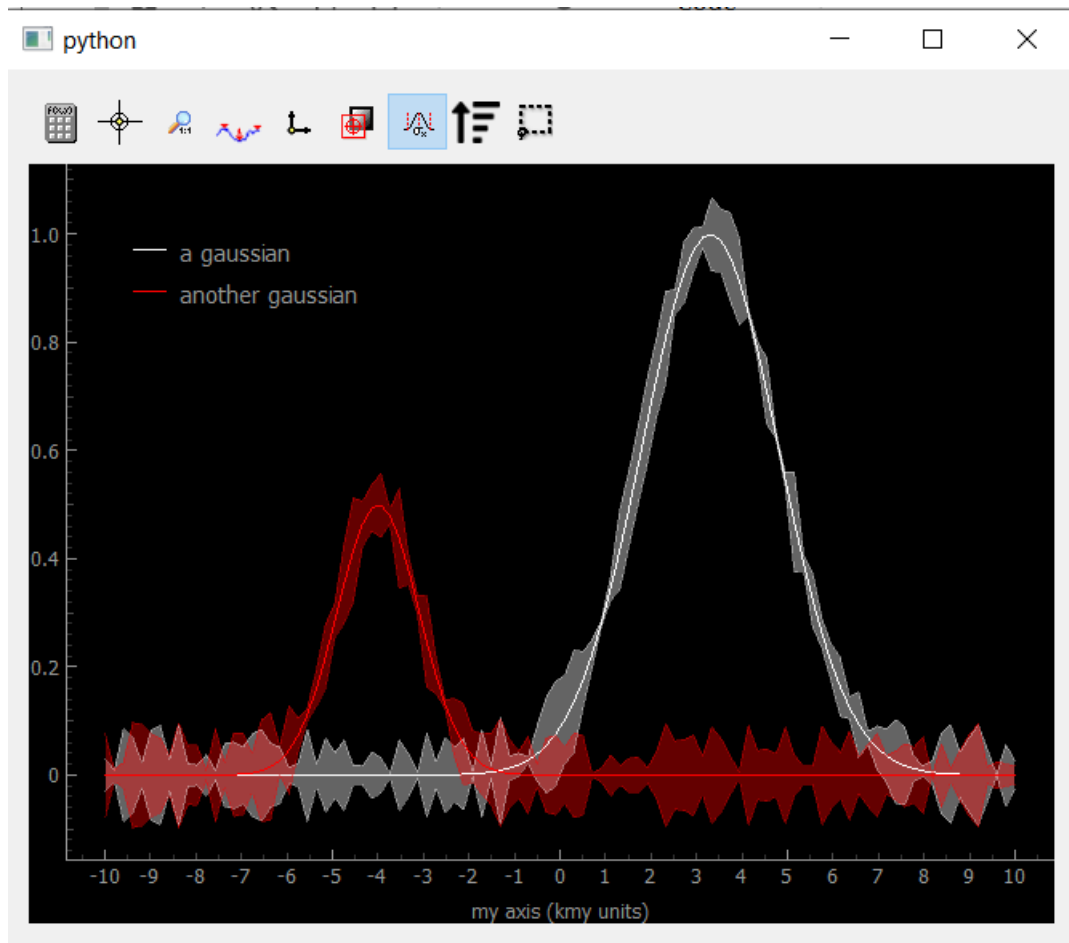


Fig. 7.77: Showing Data1D with error bars as an area around the curves.

```

axis_shuffled_array = axis.get_data()
np.random.shuffle(axis_shuffled_array)
axis_shuffled = Axis('my axis', units='my units', data=axis_shuffled_array)

dwa = DataRaw('my_1D_data', data=[mutils.gauss1D(axis_shuffled.get_data(), 3300, 2500),
                                mutils.gauss1D(axis_shuffled.get_data(), -4000, 1500)
                                * 0.5],
              labels=['a gaussian', 'another gaussian'],
              axes=[axis_shuffled])
dwa.plot('qt')

```

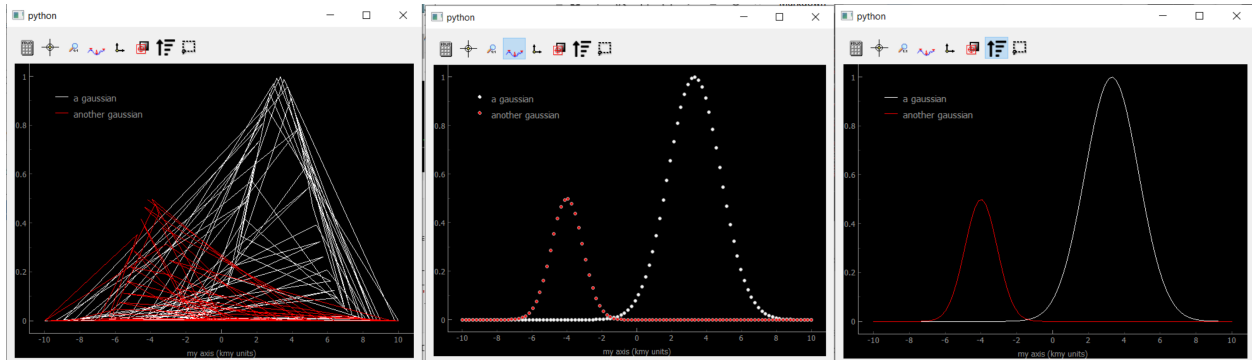


Fig. 7.78: Showing Data1D Spread. The scrambled lines (left) still represents Gaussians, it is just that the random ordering scrambled the lines. If one remove the lines by clicking the *dot only* button, the Gaussians reappear (middle). They reappear also after pressing the sort button (right).

Plotting 2D data

2D data can be either an image (pixels on a regular grid) or a collection of scalars with XY coordinates. PyMoDAQ introduce therefore the notion of “uniform” data for the former and “spread” data for the later. They can however be transparently plotted on the same *Viewer2D* data viewer. One will first show both cases before discussing the *Viewer2D* toolbar.

Uniform data

Let’s generate data displaying 2D Gaussian distributions:

```

# generating uniform 2D data
NX = 100
NY = 50
x_axis = Axis('xaxis', 'xunits', data=np.linspace(-20, 20, NX), index=1)
y_axis = Axis('yaxis', 'yunits', data=np.linspace(20, 40, NY), index=0)

data_arrays_2D = [mutils.gauss2D(x_axis.get_data(), -5, 10, y_axis.get_data(), 25, 2) +
                  mutils.gauss2D(x_axis.get_data(), -5, 5, y_axis.get_data(), 35, 2) * 0.
                  * 0.1,
                  mutils.gauss2D(x_axis.get_data(), 5, 5, y_axis.get_data(), 30, 8)]
data2D = DataRaw('data2DUniform', data=data_arrays_2D, axes=[x_axis, y_axis],

```

(continues on next page)

(continued from previous page)

```
labels=['red gaussian', 'green gaussian']
data2D.plot('qt')
```

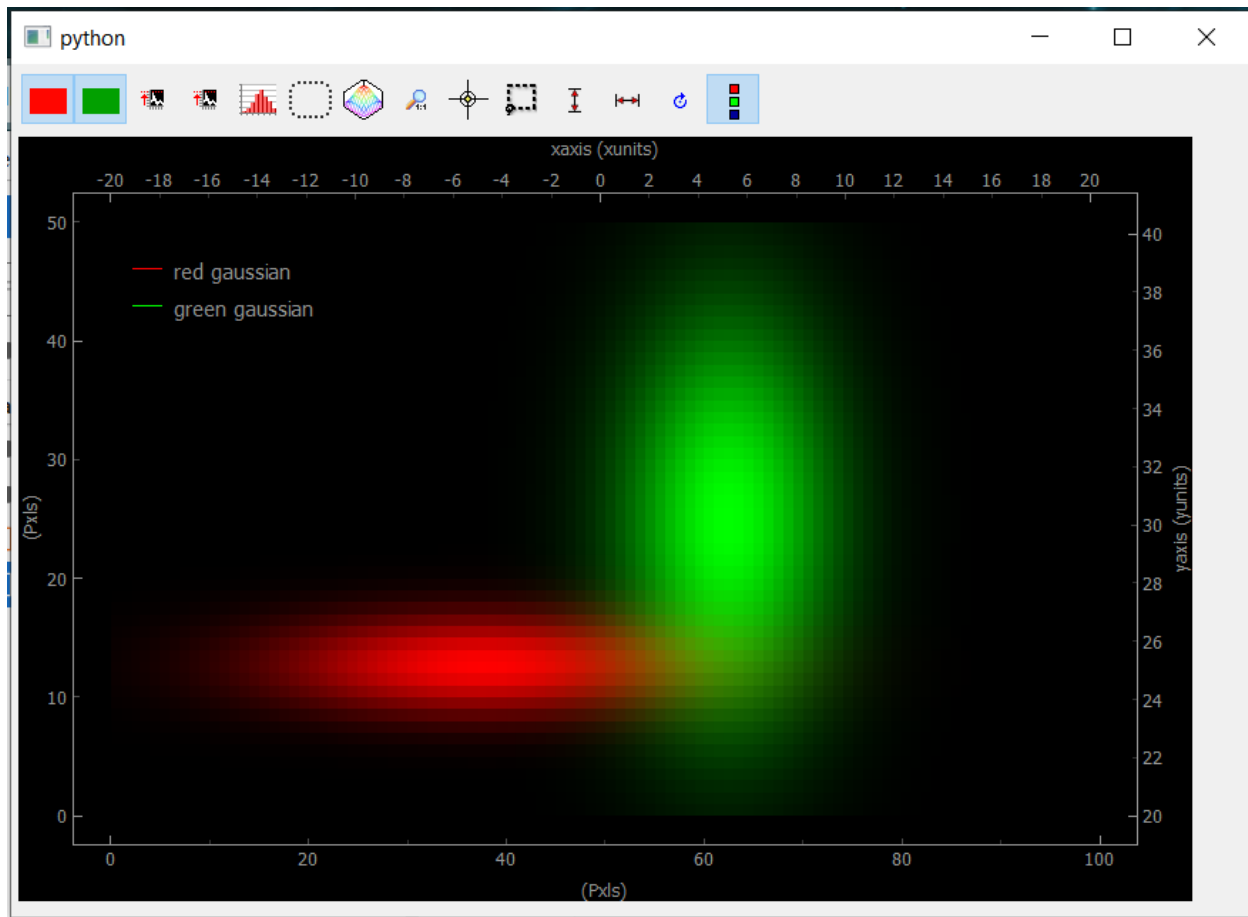


Fig. 7.79: Showing uniform Data2D

The bottom and left axes correspond to the image pixels while the right and top ones correspond to the real physical axes defined in the data object. When several arrays are included into the data object, they will be displayed as RGB layers. Data visibility can be set using the red/green (blue) buttons. If only one array is used, the color will be white.

Spread Data

Spread 2D data are typically what you get when doing a *Spread* or *Tabular* 2D scan, see [Scanner](#). By the way, *Spread* or *Tabular* 1D scan would typically give the scrambled plot on figure [Fig. 7.78](#). Let's generate and plot such 2D data

```
# generating Npts of spread 2D data
N = 100
x_axis_array = np.random.randint(-20, 50, size=N)
y_axis_array = np.random.randint(20, 40, size=N)

x_axis = Axis('xaxis', 'xunits', data=x_axis_array, index=0, spread_order=0)
y_axis = Axis('yaxis', 'yunits', data=y_axis_array, index=0, spread_order=1)
```

(continues on next page)

(continued from previous page)

```

data_list = []
for ind in range(N):
    data_list.append(mutils.gauss2D(x_axis.get_data()[ind], 10, 15,
                                   y_axis.get_data()[ind], 30, 5))
data_array = np.squeeze(np.array(data_list))

data2D_spread = DataRow('data2DSpread', data=[data_array],
                        axes=[x_axis, y_axis],
                        distribution='spread',
                        nav_indexes=(0,))
data2D_spread.plot('qt')

```

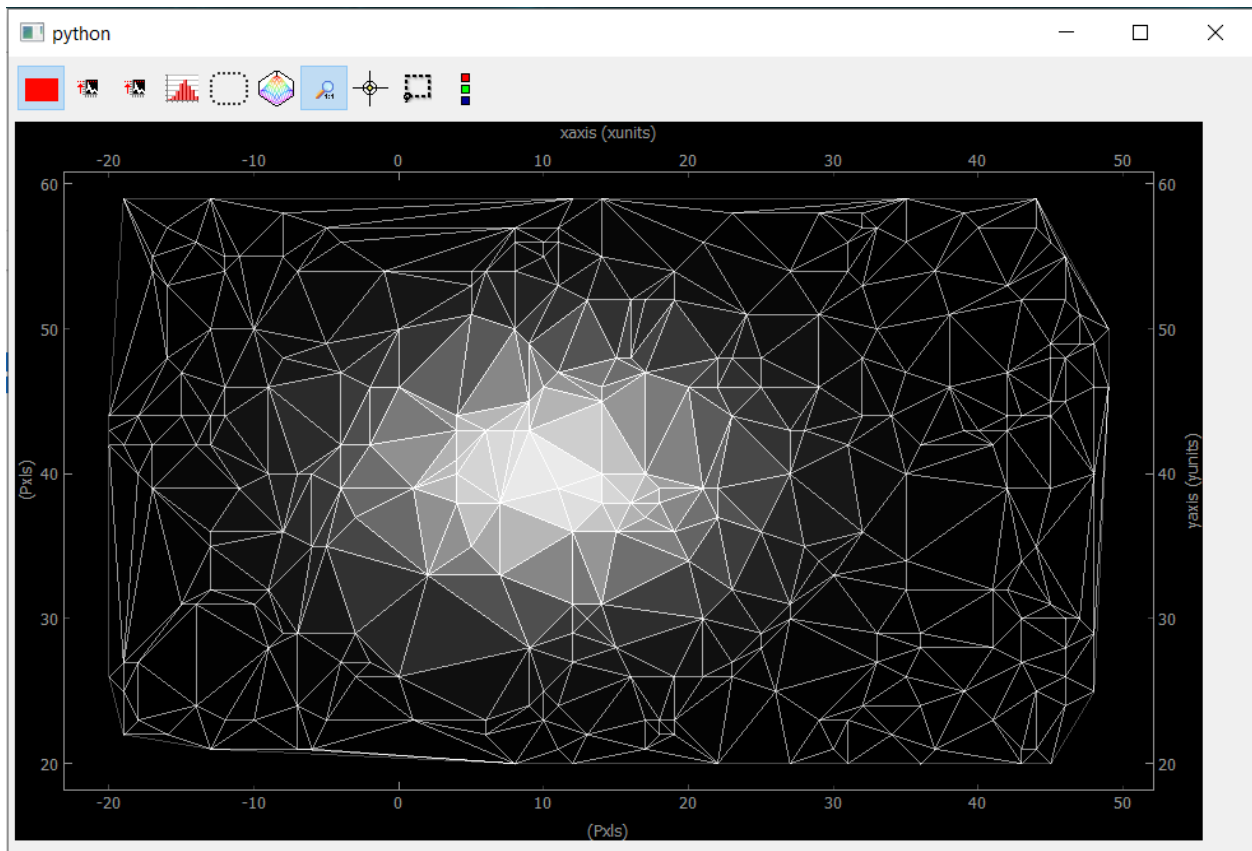


Fig. 7.80: Showing Data2D Spread. Each point in the spread collection is a vertex in the mesh while the color of the triangle is given by the mean of the three vertex.

If we go back to the construction of the data object, you may have noticed the introduction of a *nav_indexes* parameter and a *distribution* parameter. The latter is usually and by default equal to *uniform* but here we have to specify that the data will be a collection of *spread* points.

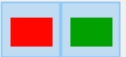







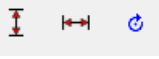

By construction, *spread* data have navigation axes, the coordinates of the points (note that the scalar *points* in our example could also be *Data1D* or *Data2D* points, we'll see that with the *ViewerND*) and specifying the distribution to *spread* allows PyMoDAQ to handle this properly compared to the *uniform* case.

But then, the parameter *nav_indexes* is used to specify which dimension of the data array will be considered navigation, the rest being signal. However in our collection, the shape of the data is only (100,) so *nav_indexes* is (0,). But still,

we do have two axes: the X and Y coordinates of our points... To handle this, the Axis object has to include a new parameter, the *spread_order* specifying which axis corresponds to which coordinate but both referring to the same *navigation* dimension of the data.

Toolbar

As for the buttons in the toolbar (you can try them from the notebook):

-  : Show/Hide the corresponding data
-  : Autoscale on the color scale (between 0 and max or between -max and max)
-  : display the histogram panel, allowing manual control of the colors and color saturation. See figure below.
-  : Open the ROI manager allowing to load, save and define rectangular or elliptical regions of interest. Each of these ROI will produce *Data1D* data (lineouts by vertical and horizontal application of a mathematical function: mean, sum... along horizontal or vertical axis of the ROI) and *Data0D* by application of the same mathematical function along both axes of the ROI.
-  : shows an isocurve specified by the position of a green line on the histogram
-  : set the aspect ratio to one
-  : activate the crosshair (see figure below)
-  : extra rectangular ROI that can be used independently of the ROI manager
-  : flip or rotate the image
-  : show/hide the legend (see figure below)

On figure [Fig. 7.81](#), the histogram has been activated and we rescaled the red colorbar to saturate the red plot and make the tiny Gaussian that was hidden to appear. We also activated the crosshair that induced the plotting of *Data1D* (taken for both channel along the crosshair lines) and *Data0D* (at the crosshair position and plotted on the bottom right).

Plotting all other data

All data that doesn't fit the explanations above should be plotted using the *ViewerND*. This viewer is a combination of several *Viewer0D*, *Viewer1D* and *Viewer2D* allowing to plot almost any kind of data. The figure below shows the basic look of the *ViewerND*. It consists in a Navigation panel and a Signal panel, dealing with the notion of signal/navigation, see [DataND](#).

```
from pymodaq.utils.plotting.data_viewers.viewerND import ViewerND
viewerND = ViewerND()
```

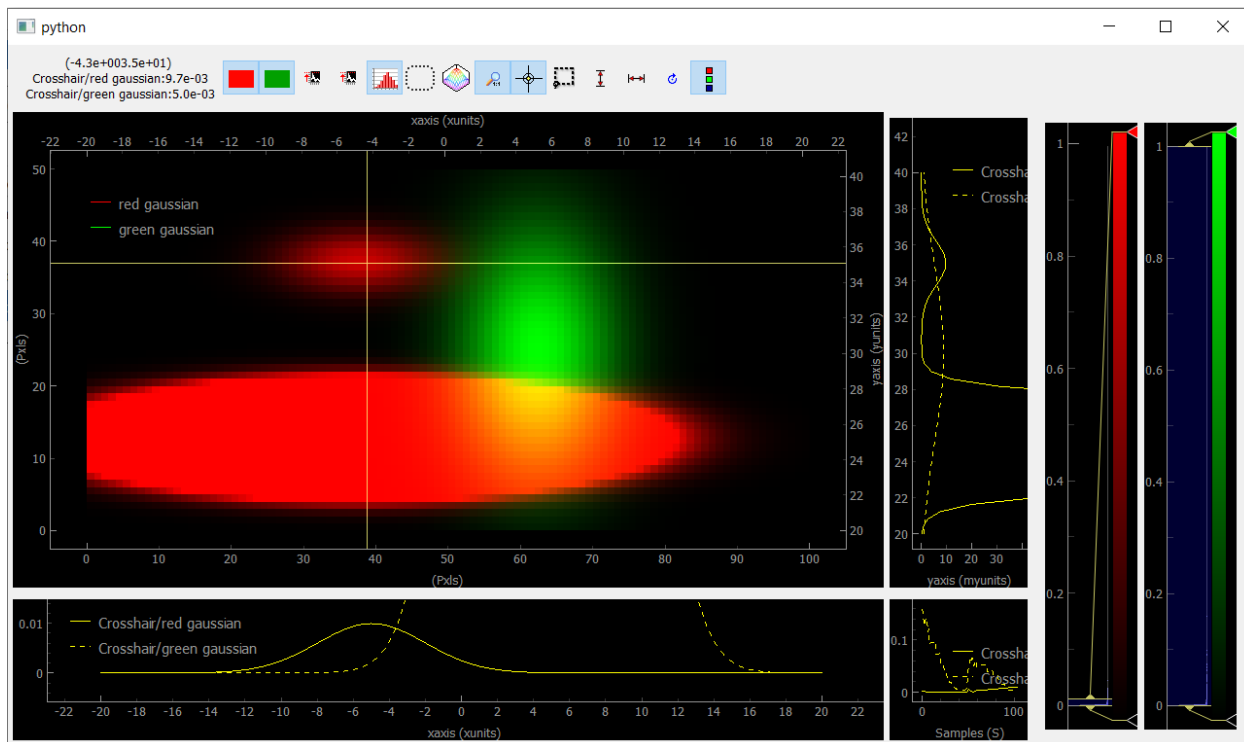
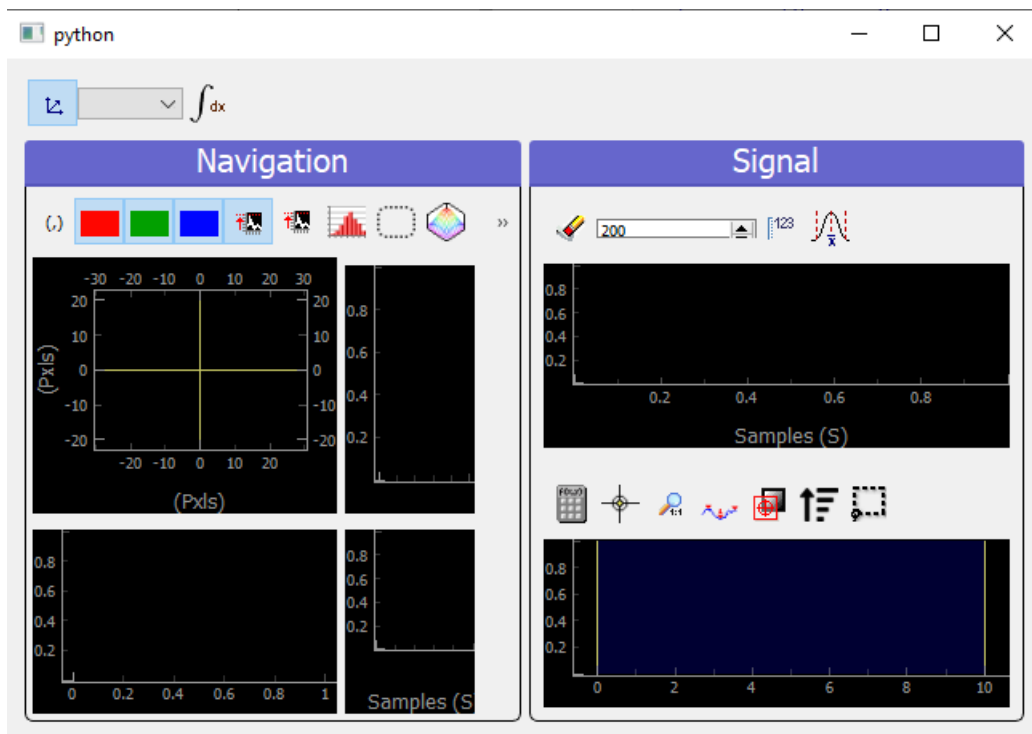


Fig. 7.81: Viewer2D with toolbar buttons activated and image saturation from the histogram.

Fig. 7.82: An empty *ViewerND*

Not much yet to say about it, but let's load some complex data and plot it with this viewer. For the first example, we'll get tomographic data (3D) from the human brain. We'll get that from the *Statistical Parametric Mapping* software website hosted [here](#).

```
import tempfile
from pathlib import Path
import zipfile
from urllib.request import urlretrieve
import nibabel

# Create a temporary directory
with tempfile.TemporaryDirectory() as directory_name:
    directory = Path(directory_name)
    # Define URL
    url = 'http://www.fil.ion.ucl.ac.uk/spm/download/data/attention/attention.zip'

    # Retrieve the data, it takes some time
    fn, info = urlretrieve(url, directory.joinpath('attention.zip'))

    # Extract the contents into the temporary directory we created earlier
    zipfile.ZipFile(fn).extractall(path=directory)

    # Read the image
    struct = nibabel.load(directory.joinpath('attention/structural/nsM00587_0002.hdr'))

    # Get a plain NumPy array, without all the metadata
    array_3D = struct.get_fdata()

dwa3D = DataRaw('my brain', data=array_3D, nav_indexes=(2,))
dwa3D.create_missing_axes()

viewerND.show_data(dwa3D) # or just do dwa3D.plot('qt')
```

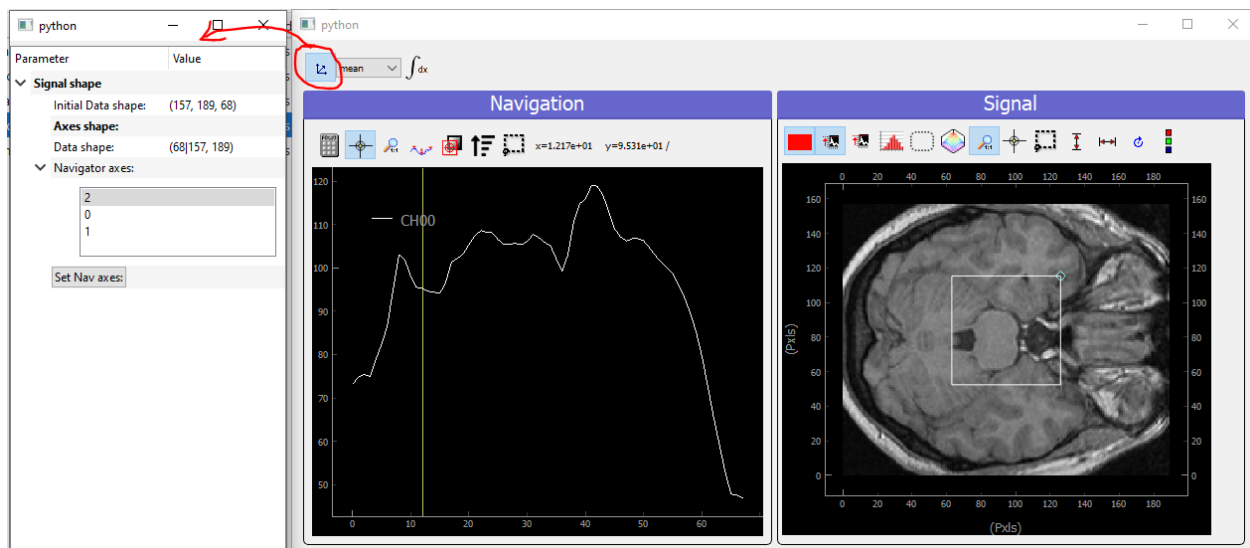

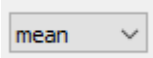



Fig. 7.83: Showing brain 3D data on a ViewerND

Here you now see the image of the brain (signal part) at a certain height (12.17, navigation part) within the skull. The

signal data is taken at the height corresponding to the crosshair vertical line within the navigation panel. Moving it laterally will display a different brain z-cut. The navigation 1D plot is calculated from the white ROI on the signal panel, applying the mathematical function to it (here *mean* see on top of the plot) and displaying this for all z-cut on the navigation panel. Therefore, moving this ROI will change the printed navigation plot. Another widget (on the left) displays information on the data: its shape and navigation/signal dimensions. From this, one can also change which axes are navigation (here this is axis 2 as specified when the data object has been constructed). In the notebook, you can change this, selecting one, two or even the three indexes and see how it's impacting on the *ViewerND*.

Some buttons in the toolbar can be used to better control the data exploration:

-  : opens a side window to control navigation axes
-  : select which mathematical operator to apply to the signal ROI in order to plot meaningful navigation data
-  : if activated, another signal plot will be generated depicting not the data indexed at the position of the crosshair but integrated over all navigation axes

Signal data dimension cannot exceed 2, meaning you can only plot signal that are *Data0D*, *Data1D* or *Data2D* which make sense as only this kind of data are produced by usual detectors. On the navigation side however, one can have as many navigation axes as needed. Below you'll see some possibilities.

Uniform Data

Let's first create a 4D Data object, we'll then see various representations as a function of its navigation indexes

```
x = mutils.linspace_step(-10, 10, 0.2)
y = mutils.linspace_step(-30, 30, 1)
t = mutils.linspace_step(-100, 100, 2)
z = mutils.linspace_step(0, 50, 0.5)

data = np.zeros((len(y), len(x), len(t), len(z)))
amp = np.ones((len(y), len(x), len(t), len(z)))
for indx in range(len(x)):
    for indy in range(len(y)):
        data[indy, indx, :, :] = amp[indy, indx] * (
            mutils.gauss2D(z, 0 + indx * 1, 20,
                          t, 0 + 2 * indy, 30)
            + np.random.rand(len(t), len(z)) / 5)

dwa = DataRaw('NDdata', data=data, dim='DataND', nav_indexes=(0, 1),
              axes=[Axis(data=y, index=0, label='y_axis', units='yunits'),
                    Axis(data=x, index=1, label='x_axis', units='xunits'),
                    Axis(data=t, index=2, label='t_axis', units='tunits'),
                    Axis(data=z, index=3, label='z_axis', units='zunits')])

dwa.plot('qt')
```

We use here (but it's done automatically from the metadata) two *Viewer2D* to plot both navigation and signal data. If we increase the number of navigation axes, it is no more possible to use the same approach.

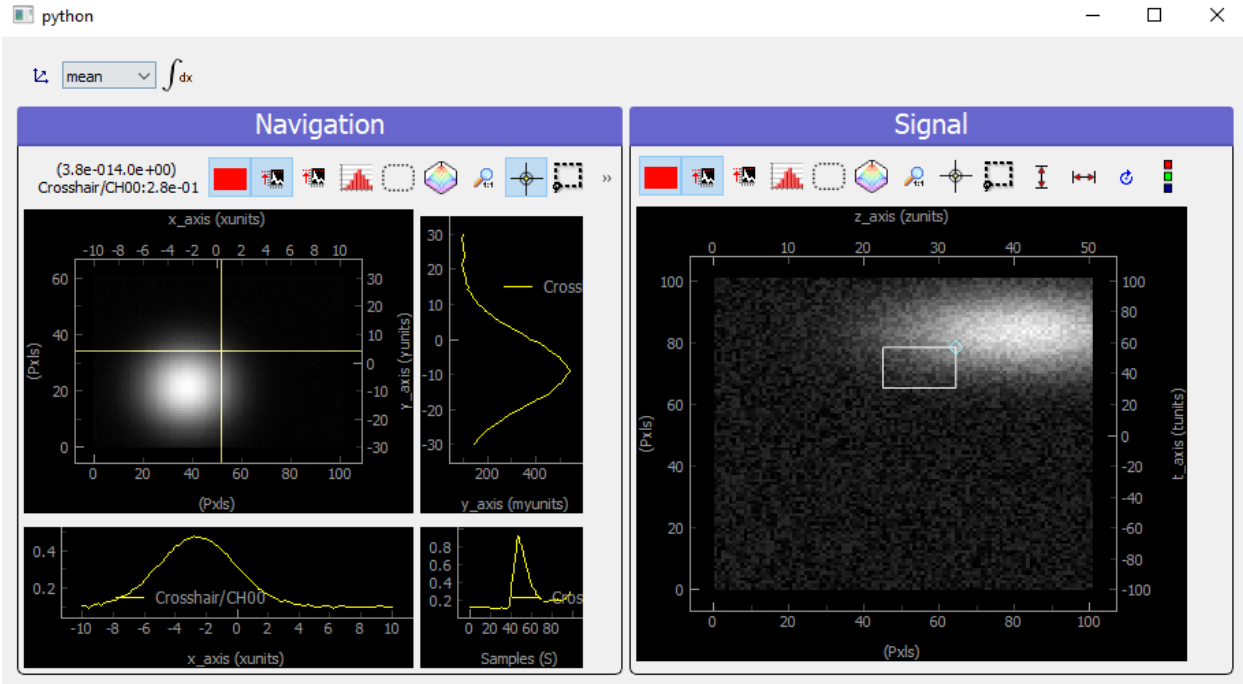


Fig. 7.84: Showing 4D uniform data on a ViewerND with two navigation axes

```
dwa.nav_indexes = (0, 1, 2)
dwa.plot('qt')
```

In that case where there are three (it could be any number >2) navigation axes. Each axis is plotted into a dedicated viewer together with a vertical yellow line allowing to index (and slice) data at this position, updating accordingly the depicted signal data

Spread Data

For *Spread* data, things are different because all navigation axes have the same length (they are the ND-coordinates of the signal data), they can therefore be plotted into the same *Viewer1D*:

```
N = 100

x = np.sin(np.linspace(0, 4 * np.pi, N))
y = np.sin(np.linspace(0, 4 * np.pi, N) + np.pi/6)
z = np.sin(np.linspace(0, 4 * np.pi, N) + np.pi/3)

Nsig = 200
axis = Axis('signal axis', 'signal units', data=np.linspace(-10, 10, Nsig), index=1)
data = np.zeros(N, Nsig)
for ind in range(N):
    data[ind,:] = mutils.gauss1D(axis.get_data(), 5 * np.sqrt(x[ind]**2 + y[ind]**2 +
    ↪ z[ind]**2) - 5, 2) + 0.2 * np.random.rand(Nsig)

dwa = DataRaw('NDdata', data=data, distribution='spread', dim='DataND', nav_indexes=(0,),
```

(continues on next page)

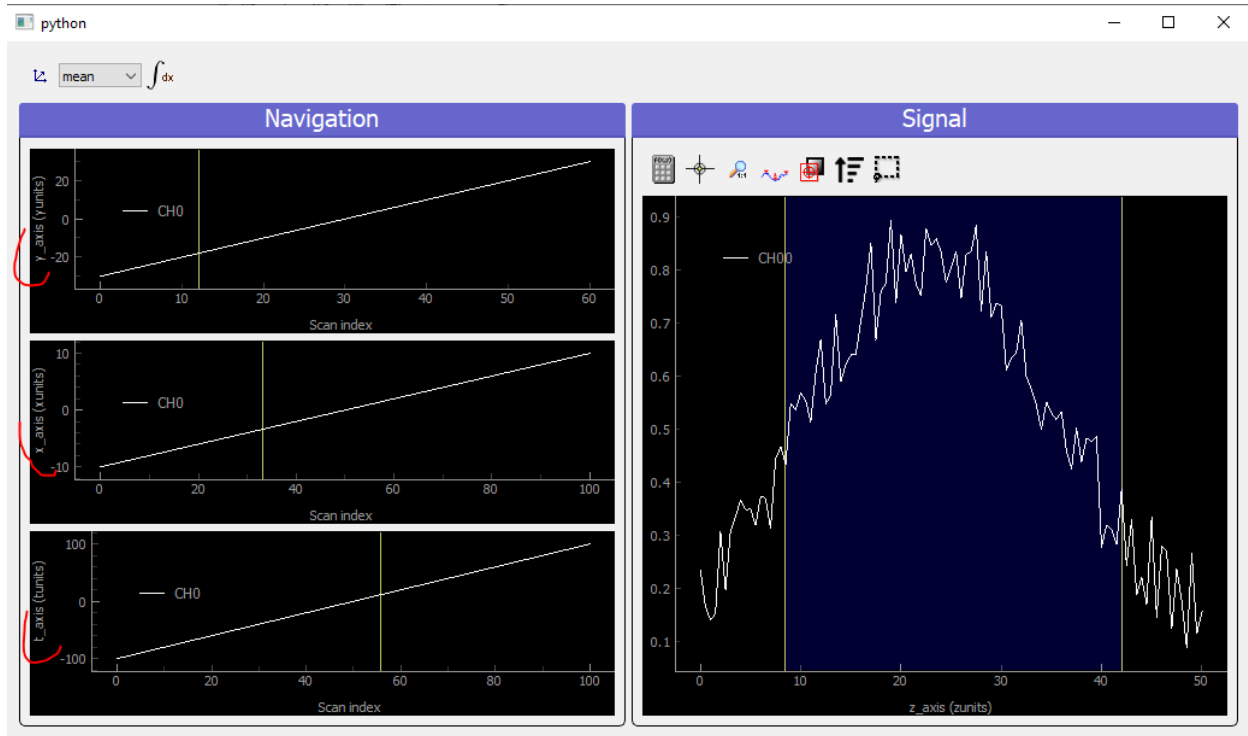


Fig. 7.85: Showing 4D uniform data on a ViewerND with three navigation axes

(continued from previous page)

```
axes=[Axis(data=x, index=0, label='x_axis', units='xunits', spread_
↪order=0),
      Axis(data=y, index=0, label='y_axis', units='yunits', spread_
↪order=0),
      Axis(data=z, index=0, label='z_axis', units='zunits', spread_
↪order=0),
      axis])

dwa.plot('qt')
```

In that case, the navigation panel is showing on the same *Viewer1D* all navigation *spread* axes (coordinates), while the signal panel shows the signal data at the index corresponding to the yellow line.

Plotting multiple data object: *ViewerDispatcher*

In PyMoDAQ, mixed data are often generated, for instance when using ROI on 2D data, lineouts (*Data1D*) will be generated as well as *Data0D*. A dedicated object exists to handle them: the *DataToExport* or *dte* in short. Well if such an object exists, a dedicated plotter should also exist, let's see:

```
from pymodaq.utils.data import DataToExport

dte = DataToExport('MyDte', data=[dwa1D, dwa3D])
dte.plot('qt')
```

Such an object is a *ViewerDispatcher*:

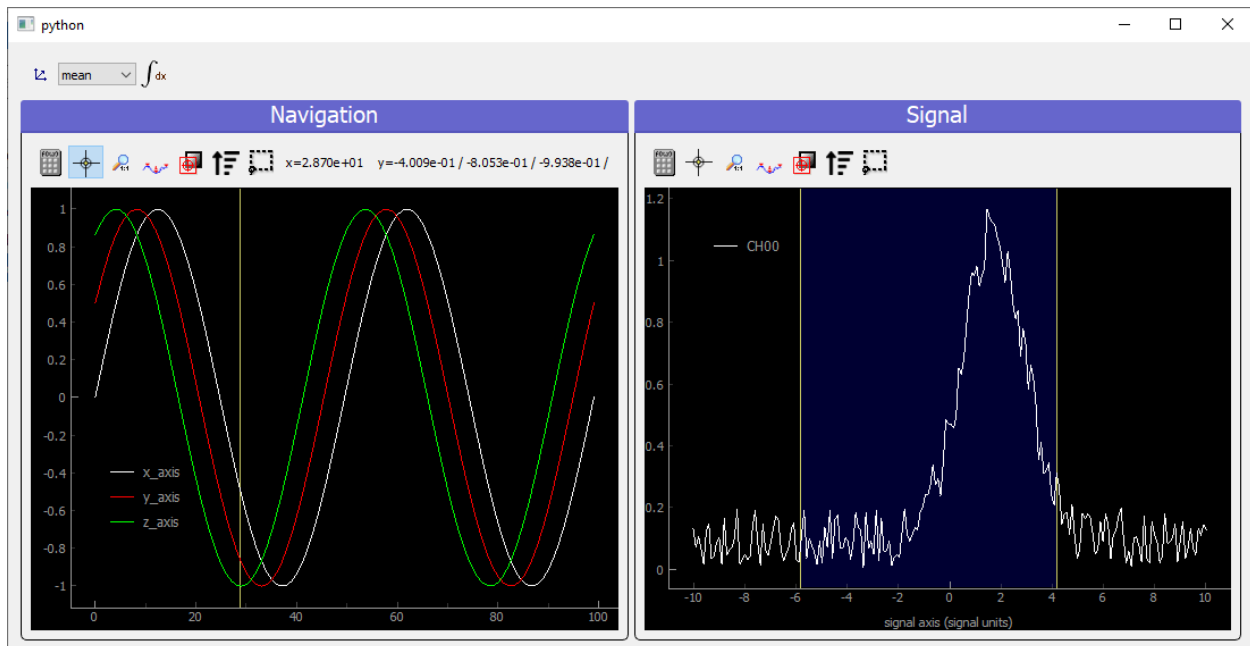


Fig. 7.86: Showing 4D spread data on a ViewerND

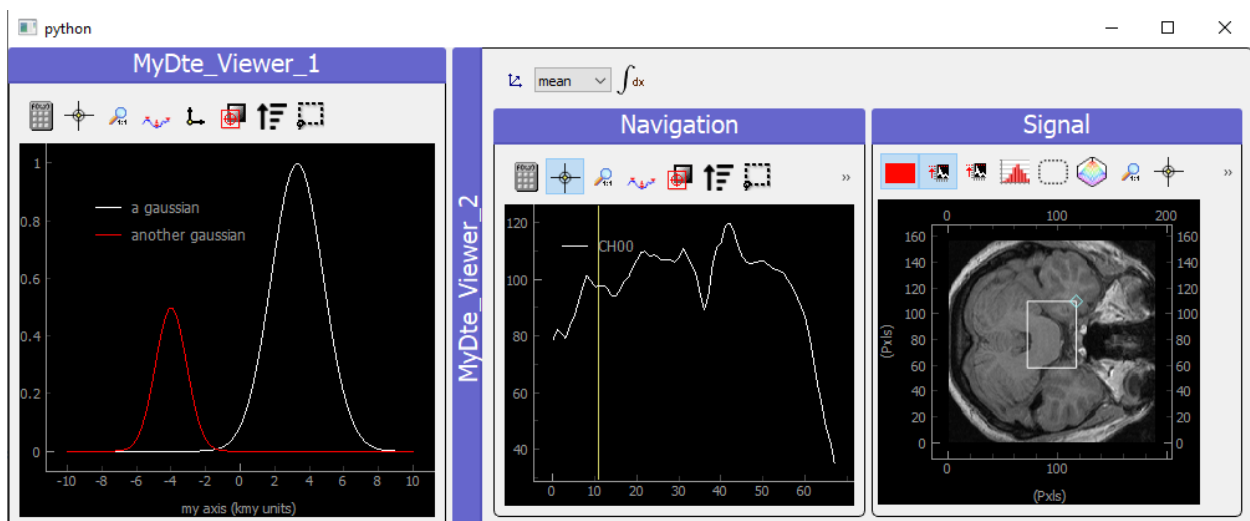


Fig. 7.87: Showing DataToExport on a ViewerDispatcher

```
from pymodaq.utils.plotting.data_viewers.viewer import ViewerDispatcher
```

It allows to generate on the fly *Docks* containing a data viewers adapted to the particular dwa it contains. Such a dispatcher is used by the *DAQ_Viewer* and the *DAQ_Scan* to display your data!

7.3.7 Useful Modules

Introduction

Utility modules are used within each main modules of PyMoDAQ but can also be used as building blocks for custom application. In that sense, all *Plotting Data* and even *DAQ_Viewer* and *DAQ_Move* can be used as building blocks to control actuators and display datas in a custom application.

Module Manager

The module manager is an object used to deal with:

- Selection of actuators and detectors by a user (and internal facilities to manipulate them, see the API when it will be written...)
- Synchronize acquisition from selected detectors
- Synchronize moves from selected actuators
- Probe as lists all the datas that will be exported by the selected detectors (see [Fig. 7.88](#))
- Test Actuators positioning. Clicking on test_actuator will let you enter positions for all selected actuators that will be displayed when reached

Scan Selector

Scans can be specified manually using the *Scanner Settings* (explained above). However, in the case of a scan using 2 *DAQ_Move* modules, it could be more convenient to select an area using a rectangular ROI within a 2D viewer. Various such viewers can be used. For instance, the viewer of a camera (if one think of a camera in a microscope to select an area to cartography) or even the *DAQ_Scan* 2D viewer. Sometimes it could also be interesting to do linear sections within a 2D phase space (let's say defined by the ranges of 2 *DAQ_Moves*). This defines complex tabular type scan within a 2D area, difficult to set manually. [Fig. 7.38](#) displays such sections within the *DAQ_Scan* viewer where a previous 2D scan has been recorded. The user just have to choose the correct *selection* mode in the *scanner settings*, see [Fig. 7.89](#), and select on which 2D viewer to display the ROI (*From Module* option).

Module Manager

This module is made so that selecting actuators and detectors for a given action is made easy. On top of it, there are features to test communication and retrieve infos on exported datas (mandatory from the adaptive scan mode) or positioning. Internally, it also features a clean way to synchronize detectors and actuators that should be set together within a single action (such as a scan step).



Parameter	Value
▼ Actuators/Detectors Selection	
▼ detectors	<div> Det 1D Det 0D Det 2D </div>
▼ Actuators	<div> Theta Axis Yaxis Xaxis </div>
Moves done?	<input checked="" type="radio"/> 
Detections done?	<input checked="" type="radio"/> 
▼ Data dimensions	
	probe_data
▼ Data0D list:	<div> Det 0D/CH000 Det 1D/Measure_000 Det 1D/Measure_001 </div>
▼ Data1D list:	<div> Det 1D/CH000 Det 1D/CH001 </div>
> Data2D list:	
> DataND list:	
▼ Actuators positions	
	test_actuator
▼ Positions:	<div> Theta Axis: 25.0 Xaxis: 54.0 Yaxis: 12.0 </div>

Fig. 7.88: The Module Manager user interface with selectable detectors and actuators, with probed data feature and actuators testing.

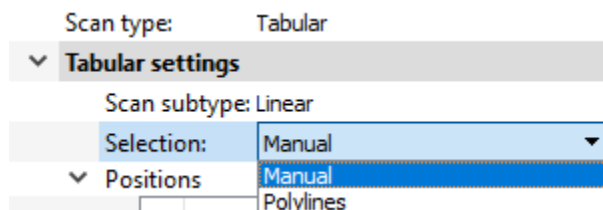


Fig. 7.89: In the scanner settings, the selection entry gives the choice between *Manual* selection of from *PolyLines* (in the case of 1D scans) or *From ROI* in the case of 2D scans.

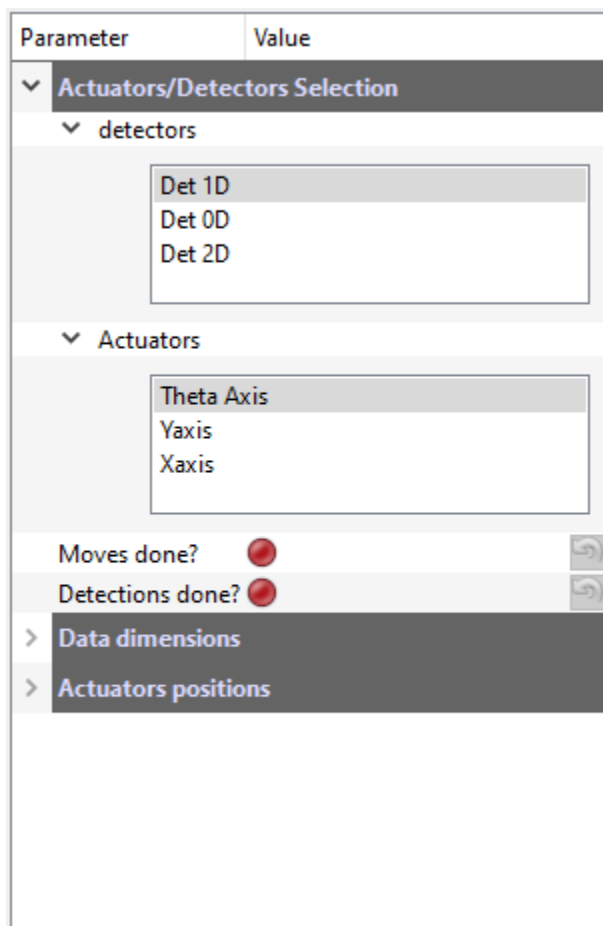


Fig. 7.90: User interface of the module manager listing detectors and actuators that can be selected for a given action.

H5Saver

This module is a help to save data in a hierarchical hdf5 binary file through the **pytables** package. Using the H5Saver object will make sure you can explore your datas with the H5Browser. The object can be used to: punctually save one set of data such as with the DAQ_Viewer (see `daq_viewer_saving_single`), save multiple acquisition such as with the DAQ_Scan (see *Saving: Dataset and scans*) or save on the fly with enlargeable arrays such as the *Continuous Saving* mode of the DAQ_Viewer.

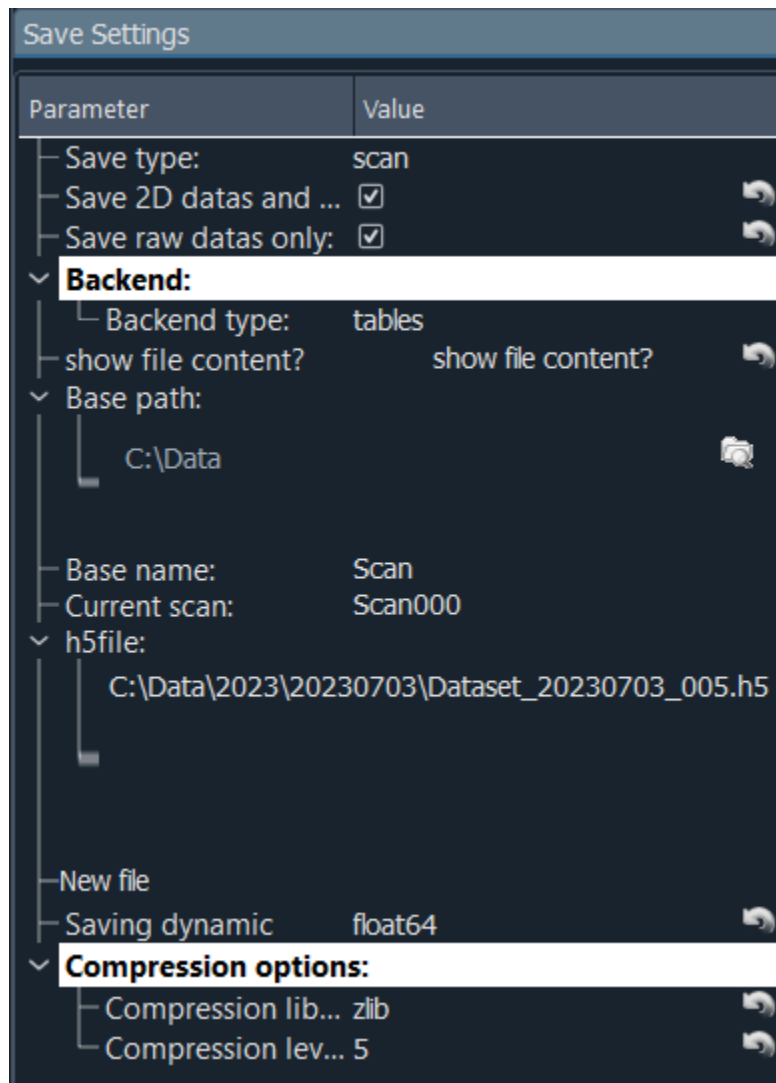


Fig. 7.91: User interface of the H5Saver module

On the possible saving options, you'll find (see Fig. 7.91):

- *Save type*:
- *Save 2D and above*: True by default, allow to save data with high dimensionality (taking a lot of memory space)
- *Save raw data only*: True by default, will only save data not processed from the Viewer's ROIs.
- *backend* display which backend is being used: pytables or h5py

- *Show file content* is a button that will open the H5Browser interface to explore data in the current h5 file
- *Base path*: where will be saved all the data
- *Base name*: indicates the base name from which the actual filename will derive
- *Current scan* indicate the increment of the scans (valid for DAQ_Scan extension only)
- *h5file: readonly*, complete path of the saved file
- *Do Save*: Initialize the file and logging can start. A new file is created if clicked again, valid for the continuous saving mode of the DAQ_Viewer
- *New file* is a button that will create a new file for subsequent saving
- *Saving dynamic* is a list of number types that could be used for saving. Default is float 64 bits, but if your data are 16 bits integers, there is no use to use float, so select int16 or uint16
- *Compression options*: data can be compressed before saving, using one of the proposed library and the given value of compression [0-9], see *pytables* documentation.

Preset manager

The *Preset manager* is an object that helps to generate, modify and save preset configurations of *DashBoard*. A preset is a set of actuators and detectors represented in a tree like structure, see [Fig. 7.92](#).

Each added module load on the fly its settings so that one can set them to our need, for instance COM port selection, channel activation, exposure time... Every time a preset is created, it is then *loadable*. The *init?* boolean specifies if the *Dashboard* should try to initialize the hardware while loading the module in the dashboard.

Overshoot manager

The *Overshoot* manager is used to configure **safety actions** (for instance the absolute positioning of one or more actuators, such as a beam block to stop a laser beam) when a detected value (from a running detector module) gets out of range with respect to some predefined bounds, see [Fig. 7.93](#). It is configurable in the framework of the Dashboard module, when actuators and detectors have been activated. A file containing its configuration will be saved (with a name derived from the preset configuration name and will automatically be loaded with its preset if existing on disk)

ROI manager

The *ROI* manager is used to save and load in one click all ROIs or Lineouts defined in the current detector's viewers, see [Fig. 7.94](#). The file name will be derived from the preset configuration file, so that at start up, it will automatically be loaded, and ROIs and Lineouts will be restored.

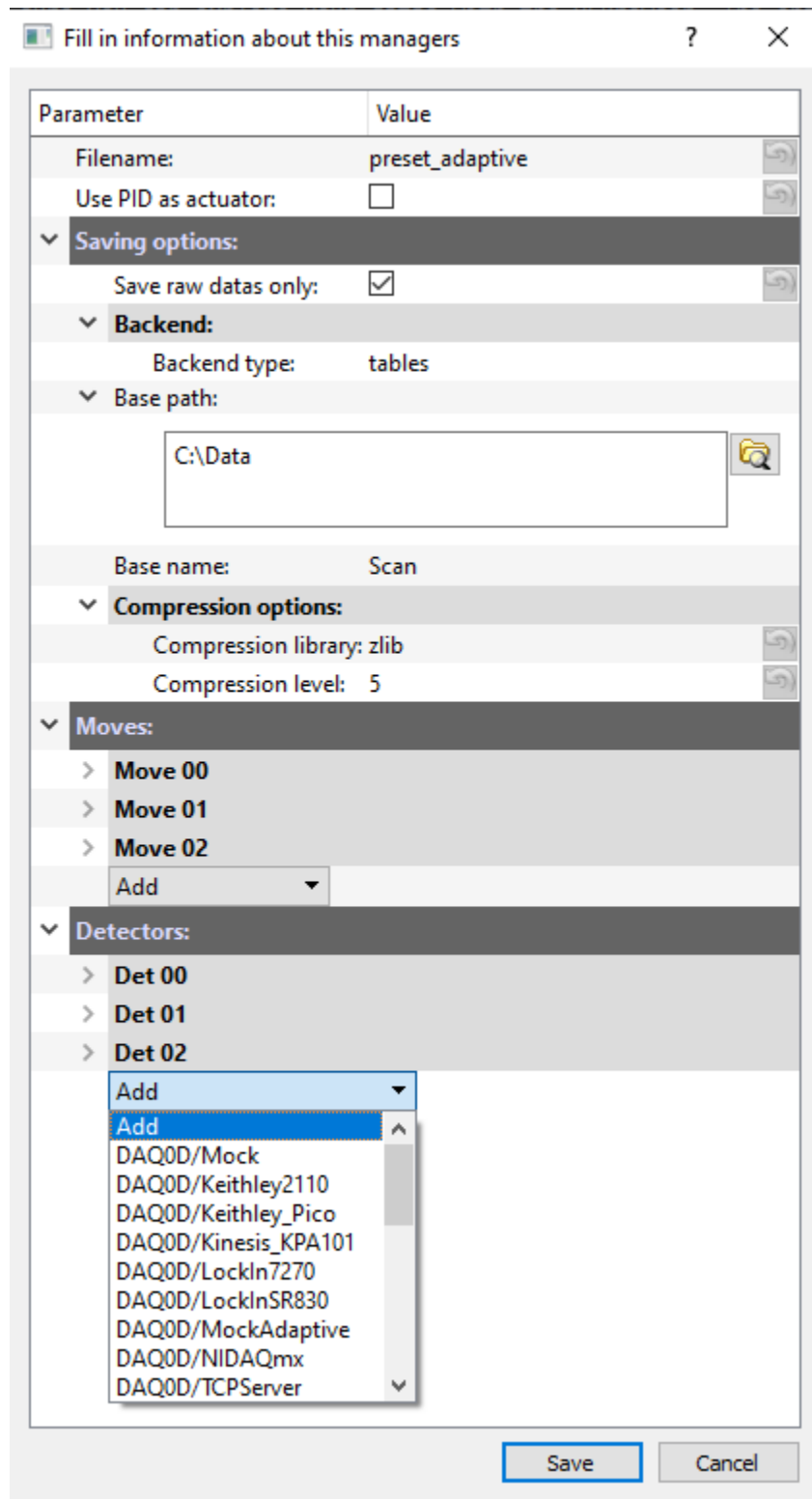


Fig. 7.92: An example of a preset creation named *preset_adaptive* containing 3 DAQ_Move modules and 3 detector modules and just about to select a fourth detector from the list of all available detector plugins.

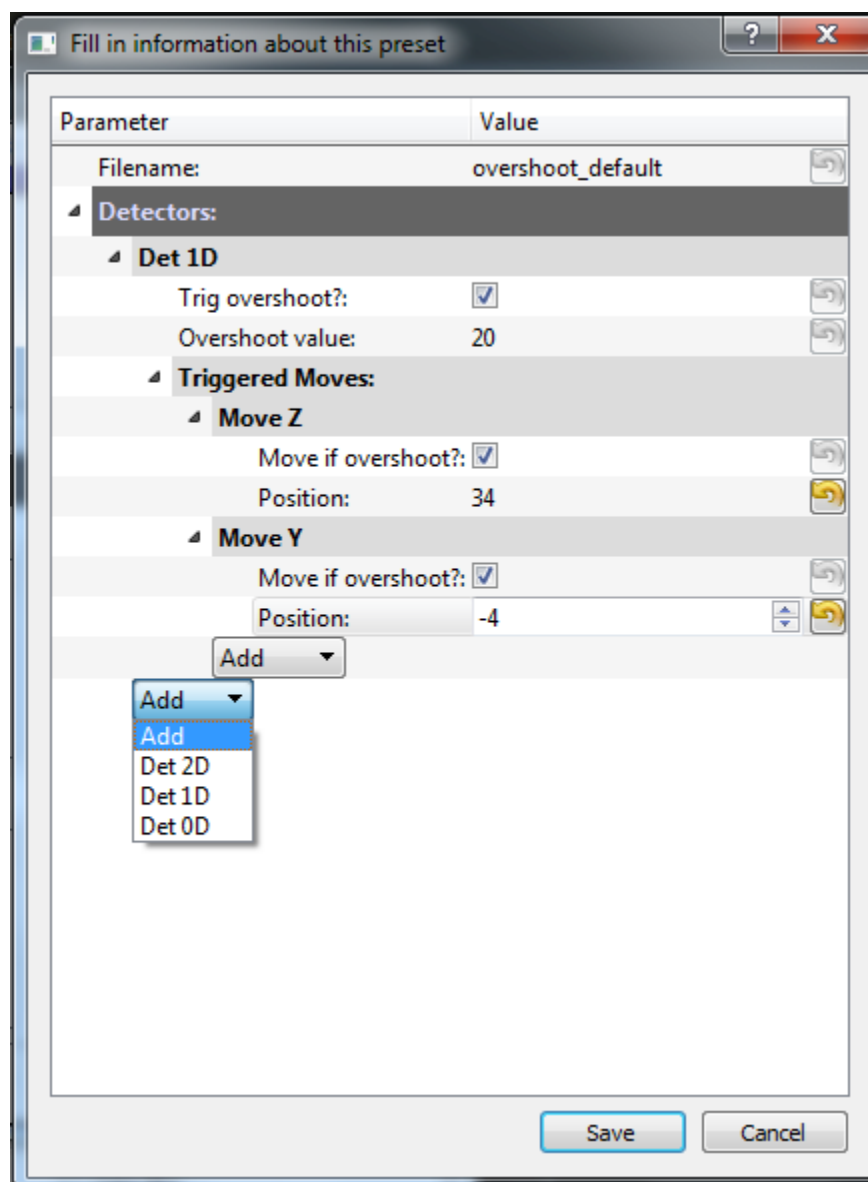


Fig. 7.93: An example of an overshoot creation named *overshoot_default* (and corresponding xml file) containing one listening detector and 2 actuators to be activated.

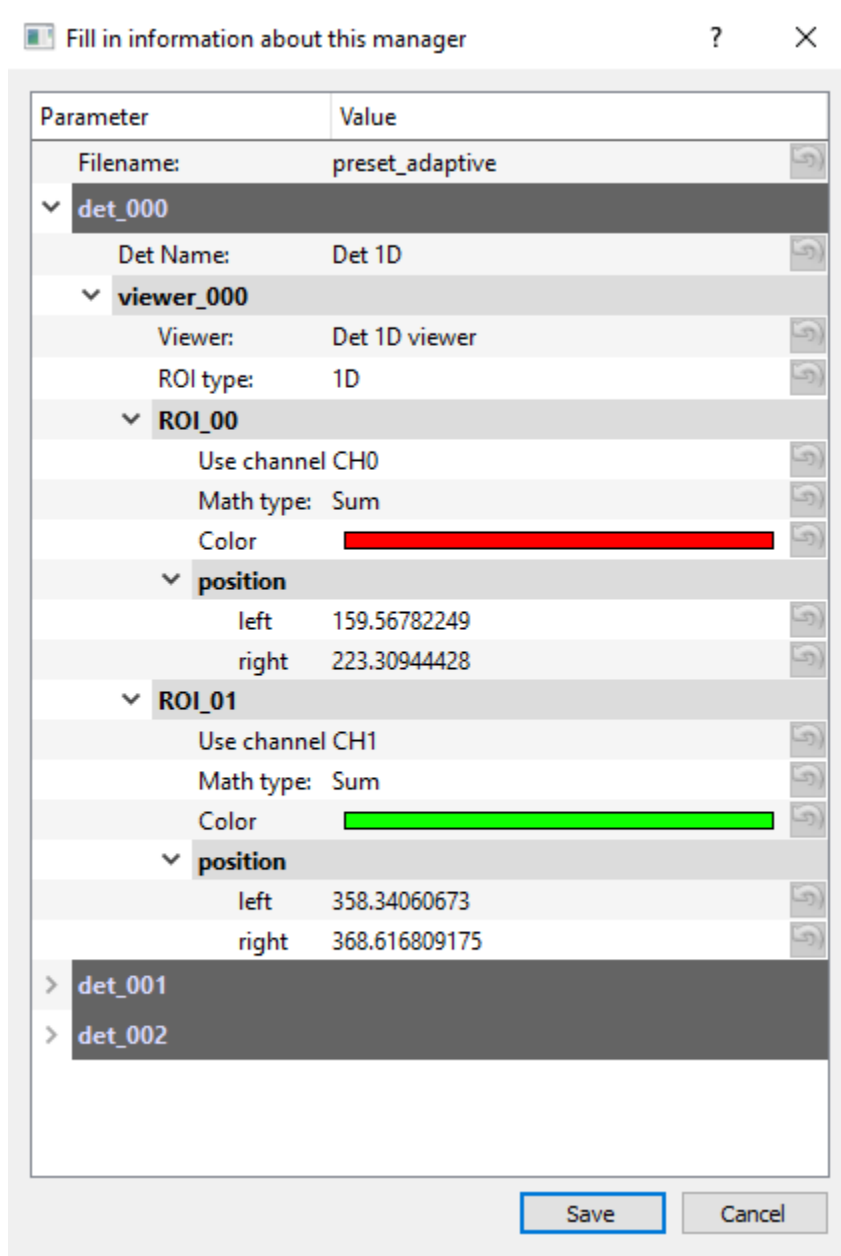


Fig. 7.94: An example of ROI manager modification named from the preset *preset_adaptive* (and corresponding xml file) containing all ROIs and lineouts defined on the detectors's viewers.

DAQ_Measurement

In construction

Navigator

See *Navigator*

Remote Manager

In construction

ChronoTimer

Fig. *User Interface of the Chrono/Timer UI* shows a user interface to be used for timing things. Not really part of PyMoDAQ but well could be useful (Used it to time a roller event in my lab ;-)

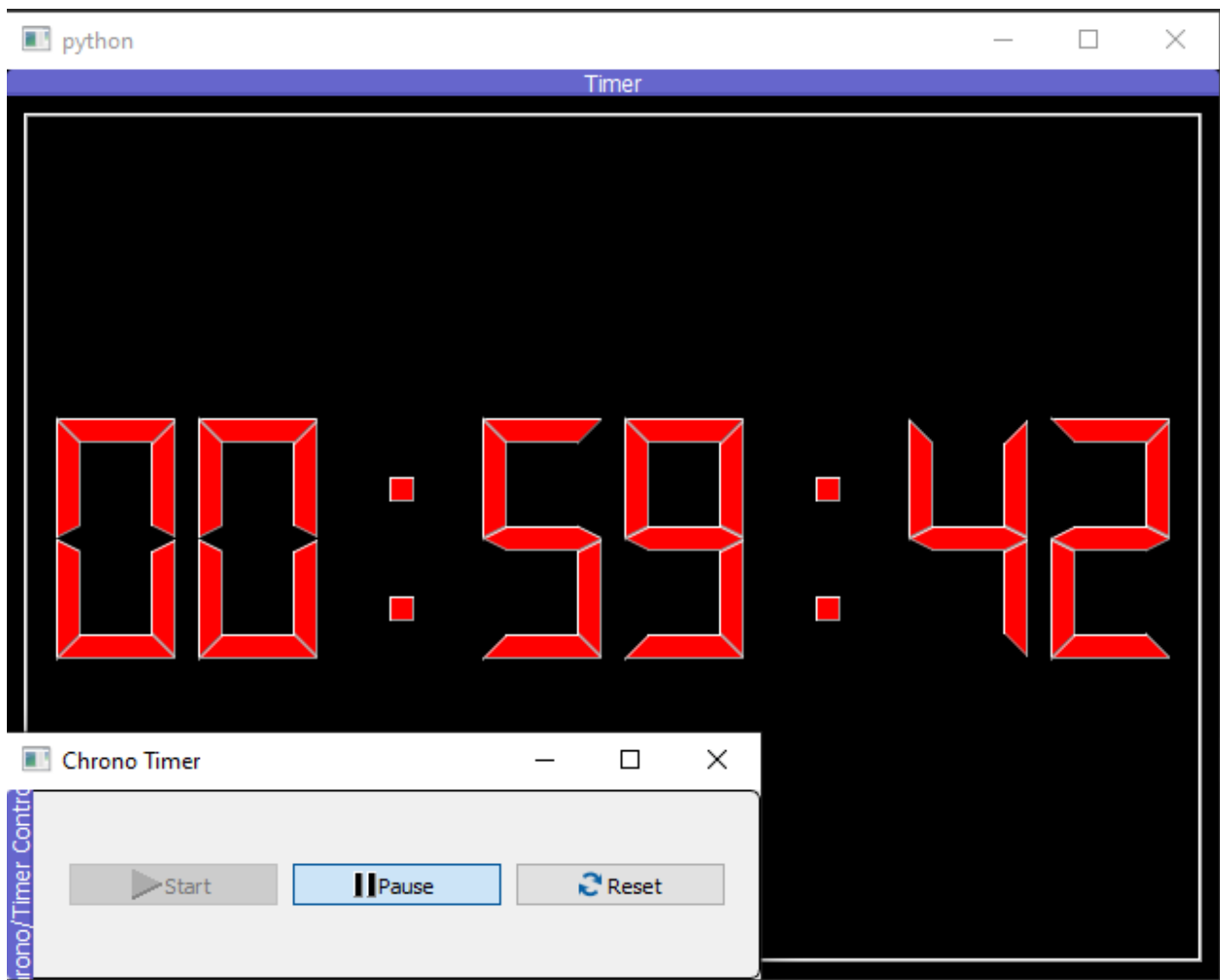


Fig. 7.95: User Interface of the Chrono/Timer UI

7.3.8 TCP/IP communication

This section is for people who want an answer to: *I have a detector or an actuator controlled on a distant computer and cannot have it on the main computer, do you have a solution?*

The answer is of course : *YES*

For this, you have two options:

- install PyMoDAQ to control your hardware on the distant computer
- Use a software on the distant computer that can use TCP/IP communication following the rules given below

With PyMoDAQ

From version 1.6.0, each actuator (DAQ_Move) or detector (DAQ_View) module can be connected to their counterpart on a distant computer. For both modules, a TCPServer plugin is available and can be initialized. It will serve as a bridge between the main computer, running for instance a DAQ_Scan module, and the distant one running a usual DAQ_Move or DAQ_View module, see Fig. 7.96. Every parameter of the distant module will be exported on its server counterpart. Any modification of these parameters, either on the server or on the local module, will be updated on either the local module or the server module.

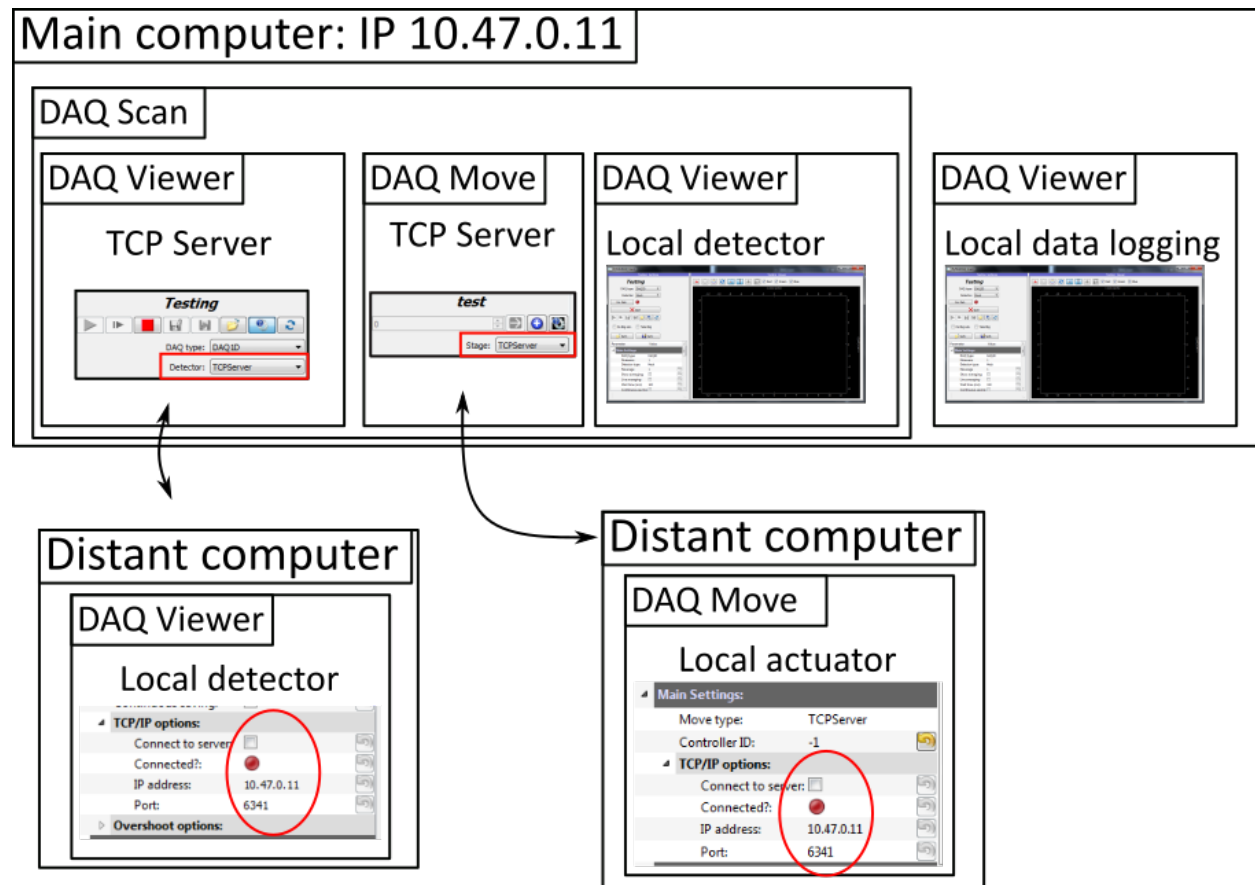


Fig. 7.96: Typical configuration with modules on distant computers communicating over a TCP/IP connection

On another software

The TCP_server plugin can also be used as a bridge between PyMoDAQ and another custom software (installed locally or on a distant computer) able to initialize a TCP client and understand PyMoDAQ's TCP/IP communications. For instance, at CEMES, we've build such a bridge between Digital Micrograph running (eventually) on a distant computer and controlling a specific Gatan camera on an electron microscope. The communication framework used by PyMoDAQ is as follow:

PyMoDAQ TCP/IP Communication protocol

Serializing objects

When dealing with TCP/IP one should first transforms object into *bytes* string (the message) and implement a mechanism to inform the client (or the server) on the length of the message. For each message (whatever the underlying object), the first 4 bytes are coding an integer whose value will be the length of the following message. Using this simple rule allow to send very complex objects.

To make sure there is a robust way to handle this in PyMoDAQ, two objects have been created, see: [ref:tcp_ip_serializer](#), respectively the *Serializer* and *DeSerializer* objects to convert a python object to bytes and from bytes to an object.

They both implements specific methods applicable to a given object but also a generic one:

```
>>> from pymodaq.utils.tcp_ip.serializer import Serializer, DeSerializer
>>> string = 'Hello'
>>> ser = Serializer(string)
>>> print(ser.string_serialization(string))
b'\x00\x00\x00\x05Hello'
```

In this example, the serializer first send 4 bytes encoding the length of the *Hello* string: *x00x00x00x05* which is the binary representation of the integer 5. Then the binary string is appended: *b'Hello*.

Similar methods exists for numbers, arrays, list, *Axis*, *DataWithAxes*...

The serialization can also be simplified using the *to_bytes()* method:

```
>>> Serializer(['Hello', 'World']).to_bytes()
b'\x00\x00\x00\x02\x00\x00\x00\x06string\x00\x00\x00\x05Hello\x00\x00\x00\x06string\x00\x00\x00\x05World'
```

Here the *list_serialization()* method has been used under the hood.

To recreate back the initial object, one should use the *DeSerializer* object:

```
>>> DeSerializer(b'\x00\x00\x00\x05Hello').string_deserialization()
Hello
>>> DeSerializer(b'\x00\x00\x00\x03<f8\x00\x00\x00\x08ffffff_@').scalar_deserialization()
125.1
```

As you see you have to know in advance which method to apply first. Therefore there is a recipe for each type of objects.

Making sure messages are complete:

Message send on a tcp/ip connection can sometimes be send as chunks, it is therefore important to know what will be the length of the message to be sent or to be received. PyMoDAQ use the following methods to make sure the message is entirely send or entirely received:

```
def check_received_length(sock,length):
    l=0
    data_bytes=b''
    while l<length:
        if l<length-4096:
            data_bytes_tmp=sock.recv(4096)
        else:
            data_bytes_tmp=sock.recv(length-l)
        l+=len(data_bytes_tmp)
        data_bytes+=data_bytes_tmp
    #print(data_bytes)
    return data_bytes

def check_sended(socket, data_bytes):
    sended = 0
    while sended < len(data_bytes):
        sended += socket.send(data_bytes[sended:])
```

Sending and receiving commands (or message):

Serializing and letting know the length of the message is not enough to recreate the initial object. One should add first a command/info on what to expect from the tcp/ip pipe. Depending on the value of this message the application know what deserialization to apply.

The PyMoDAQ client/server control modules are using specific commands as strings that should be either:

- **Client receiving messages:**

- For all modules: Info, Infos, Info_xml, set_info
- For a detector: Send Data 0D, Send Data 1D, Send Data 2D
- For an actuator: move_abs, move_home, move_rel, check_position, stop_motion

- **Client sending messages:**

- For all modules: Quit, Done, Info, Infos, Info_xml
- For a detector: x_axis, y_axis
- For an actuator: position_is, move_done

The principles of communication within PyMoDAQ are summarized on figure [Fig. 7.97](#) and as follow:

To be send, the string is converted to bytes. The length of this converted string is then computed and also converted to bytes. The converted length is first send through the socket connection and then the converted command is also sent.

For the message to be properly received, the client listen on the socket. The first bytes to arrive represent the length of the message (number of bytes).

For the detail of the python utility functions used to convert, send and receive data see [TCP/IP related methods](#).

Sending messages (or commands):

```
cmd = 'Hello world'
cmd_b = b'Hello world' #L=12
L_b = b'\x00\x00\x00\x0c' # integer on 4 bytes
```

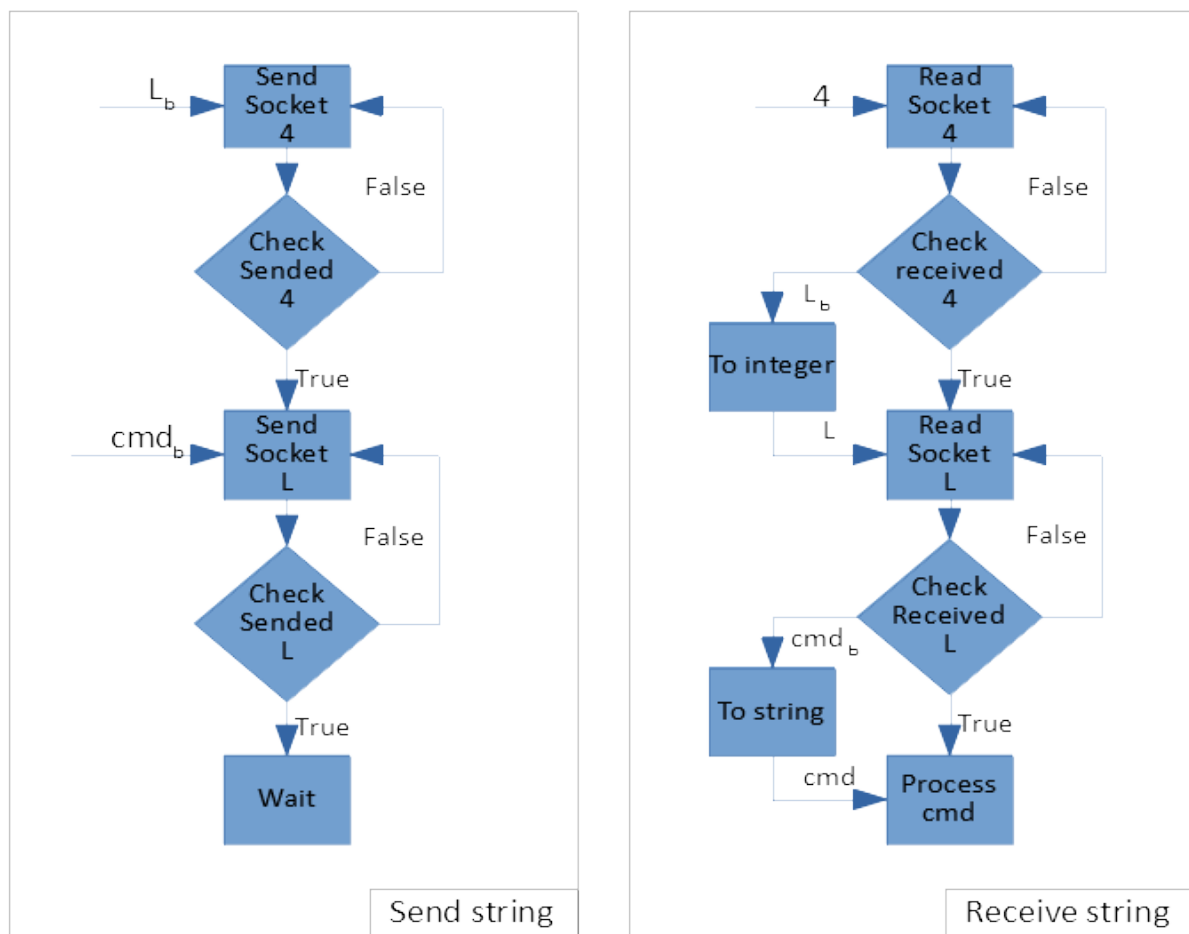


Fig. 7.97: Diagram principle of PyMoDAQ message communication through a TCP/IP socket.

Sending and receiving Datas:

Sending or receiving datas is very similar to messages except that datas have a type (integer, float...) and have also a dimensionality: 0D, 1D, ... Moreover, the datas exported from plugins and viewers are almost always numpy arrays within a list. One should therefore take all this into consideration. Below is an example of the recipe for serializing/deserializing DataWithAxes objects:

```
def dwa_serialization(self, dwa: DataWithAxes) -> bytes:
    """ Convert a DataWithAxes into a bytes string

    Parameters
    -----
    dwa: DataWithAxes

    Returns
    -----
    bytes: the total bytes message to serialize the DataWithAxes

    Notes
    ----
    The bytes sequence is constructed as:

    * serialize the string type: 'DataWithAxes'
    * serialize the timestamp: float
    * serialize the name
    * serialize the source enum as a string
    * serialize the dim enum as a string
    * serialize the distribution enum as a string
    * serialize the list of numpy arrays
    * serialize the list of labels
    * serialize the origin
    * serialize the nav_index tuple as a list of int
    * serialize the list of axis
    """
```

and obviously the deserialization process is symmetric:

```
def dwa_deserialization(self) -> DataWithAxes:
    """Convert bytes into a DataWithAxes object

    Convert the first bytes into a DataWithAxes reading first information about the
    ↳underlying data

    Returns
    -----
    DataWithAxes: the decoded DataWithAxes
    """
    class_name = self.string_deserialization()
    if class_name not in DwaType.names():
        raise TypeError(f'Attempting to deserialize a DataWithAxes flavor but got the
    ↳bytes for a {class_name}')
    timestamp = self.scalar_deserialization()
    dwa = getattr(data_mod, class_name)(self.string_deserialization(),
```

(continues on next page)

(continued from previous page)

```
source=self.string_deserialization(),
dim=self.string_deserialization(),
distribution=self.string_deserialization(),
data=self.list_deserialization(),
labels=self.list_deserialization(),
origin=self.string_deserialization(),
nav_indexes=tuple(self.list_deserialization()),
axes=self.list_deserialization(),
)
```

And because control modules send signals with *DataToExport* objects, there is also a recipe for these.

Custom client: how to?

1. The TCP/Client should first try to connect to the server (using TCP server PyMoDAQ plugin), once the connection is accepted, it should send an identification, the `client` type (*GRABBER* or *ACTUATOR* command)
2. (optional) Then it can send some information about its configuration as an xml string following the `pymodaq.utils.parameter.ioxml.parameter_to_xml_string()` method.
3. Then the client enters a loop waiting for input from the server and is ready to read commands on the socket
4. **Receiving commands**
 - For a detector: Send Data 0D, Send Data 1D, Send Data 2D
 - For an actuator: move_abs, move_home, move_rel, check_position, stop_motion
5. Processing internally the command
6. **Giving a reply**
 - **For a detector:**
 - Send the command Done
 - Send the data as a *DataToExport* object
 - **For an actuator:**
 - **Send a reply depending on the one it received:**
 - * move_done for move_abs, move_home, move_rel commands
 - * position_is for check_position command
 - Send the position as a *DataActuator* object

Pretty easy, isn't it?

Well, if it isn't you can have a look in the example folder where a Labview based TCP client has been programmed. It emulates all the rules stated above, and if you are a Labview user, you're lucky ;-) but should really think on moving on to python with PyMoDAQ...

7.3.9 LECO communication

If you want to control a device remotely, you can use **LECO** - Laboratory Experiment Control Protocol. Alternatively, you can use *TCP/IP communication*.

For that, you need to install the `pyleco` package, for example via `pip install pyleco`.

Overview

For remote control via LECO, you need three Components:

1. An *Actor*, which controls an instrument and can do *actions* on request,
2. A *Director*, which sends commands to the Actor and requests values for the Actor,
3. A *Coordinator* which transmits messages between Actors and Directors.

Coordinator

The *Coordinator* is the necessary infrastructure for the LECO network. Therefore you should start it first.

You can start it executing *coordinator* in your terminal.

Actor

Any control module from any plugins package can be made an Actor.

1. Start the module you want to control your instrument.
2. Select in the main settings the LECO options. - *Host* name and *port* are the host name and port of the Coordinator started above.
 - If the Coordinator is on the same machine (i.e. localhost) and on the default port, you do not have to enter anything.
 - *Name* defines how this module should participate in the LECO network. If you leave it empty, the name of the module is taken.
3. Click on *connect*,
4. Now the green lamp should be lit and the Actor is ready to be used

Note: You can change the name, even after having clicked connect.

Director

For remote control, we need also the *Director* in order to direct the *Actor*. You can start the *LECODirector* module from the mock plugins package, either in standalone mode or in the dashboard.

1. Start the appropriate type of *LECODirector*, either move or a viewer type.
2. Set the *Actor name* setting to the name of the actor module you wish to control.
3. Initialize the detector/actuator.
4. Read values or control the module remotely.

Developing with LECO for PyMoDAQ

Here are some hints about the use of LECO in PyMoDAQ, that you might write your own programs.

Overview

Both, the *Actor* and the *Director* have a `pyleco.Listener` which offers some methods via [JSON-RPC](#), which is used by LECO.

The Actor offers methods to do an action like initializing a movement or requesting a data readout. After the movement or data acquisition has finished, it will call a method on some remote Component. If you want, that the Actor sends the request to your Director, you have to tell the Actor about your name via the `set_remote_name()` method.

The `pymodaq.utils.leco.director_utils` module offers director classes, which makes it easier to call the corresponding methods of the Actor.

Serialization

PyMoDAQ data objects have to be transferred between modules. The payload of LECO messages are typically JSON encoded messages. Therefore, the [Serializer](#) and [DeSerializer](#) can encode/decode the data objects to bytes. For more information about serialization see [TCP/IP communication](#). In order to make a JSON string, base64 is used. The Serializer offers the `to_b64_string()` and the DeSerializer the `from_b64_string()` method.

7.4 Developer's Guide

7.4.1 Contributing

How to contribute

If you're willing to help, there are several ways to do it:

- Use PyMoDAQ and report bug or issues using github issue tracker
- Talk about PyMoDAQ to your colleagues
- Cite PyMoDAQ in your papers
- Add your instruments in plugins (see [Instrument Plugins](#))
- Work on new features, on solving bugs or issues

For the last point, here are some pointers:

you should fork and clone the up-to-date GitHub repo: <https://github.com/PyMoDAQ> using git command line or GitHub Desktop. Then create a dedicated branch name from the change you want to work on (using git).

Finally I advise to create a dedicated conda environment for this and install PyMoDAQ's package as a developer:

- `conda create -n dev_env`
- `conda activate dev_env`
- `cd` to the location of the folder where you downloaded or cloned the repository.

- install the package as a developer using the command `pip install -e ..`

Then any change on the code will be *seen* by python interpreter so that you can see and test your modifications. Think about writing tests that will make sure your code is sound and that modification elsewhere doesn't change the expected behavior.

When ready, you can create a pull request from your code into the proper branch, as discussed in the next section.

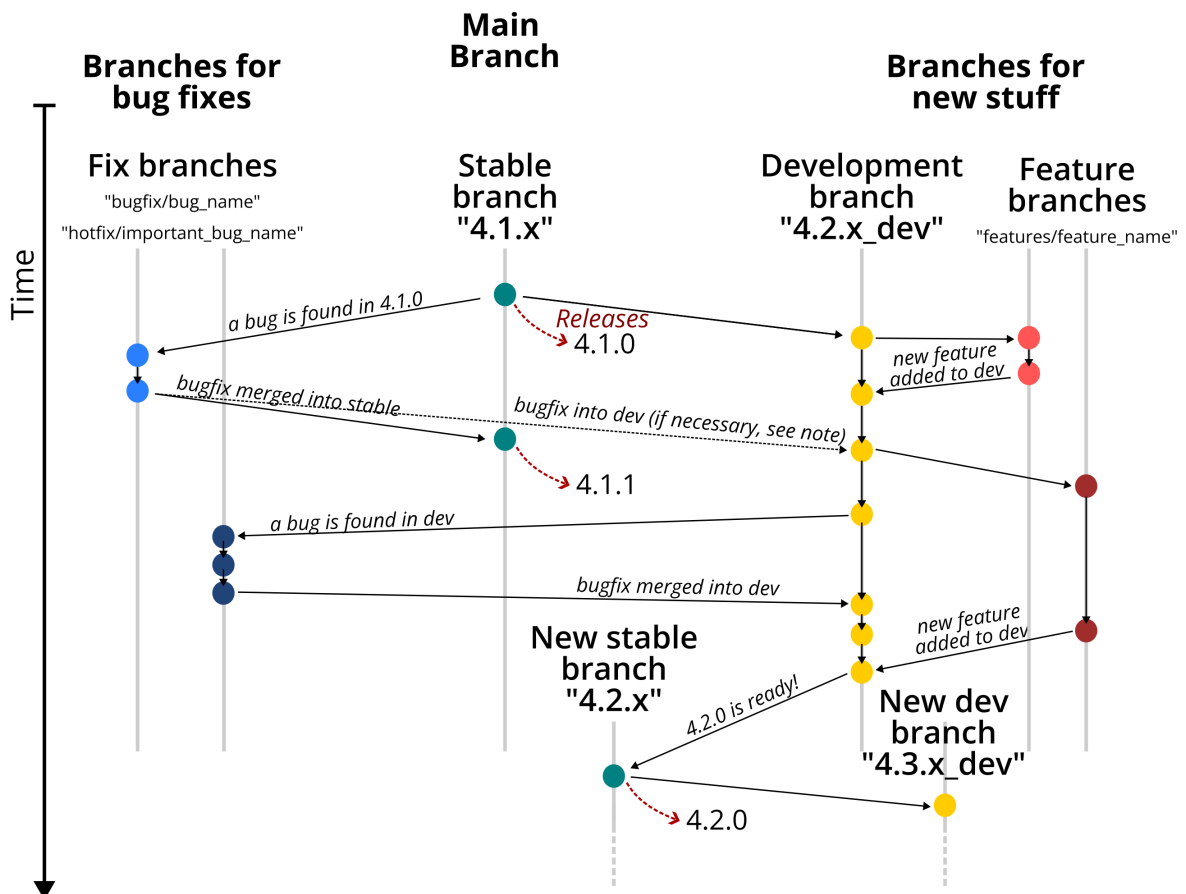
Branch structure and release cycle

There are several branches of the PyMoDAQ repository, directly linked to the *release cycle* of PyMoDAQ, which we define here. PyMoDAQ versioning follows usual practice, as described [in this link](#):

4.2.1
MAJOR Minor patch

Starting from January 2024, the following structure was agreed upon by the contributors. At any given time, there is a **stable** version of PyMoDAQ - at the time of writing it is 4.1.0 - which is not to be modified except for bugfixes, and a **development** version (currently, 4.2.0), onto which new features may be added.

The release cycle is illustrated in this figure:



This cycle makes use of several types of branches:

Code flow branches:

- **the stable branch, eg: '4.1.x'** This is the branch representing the stable version of PyMoDAQ. No change should be made on this branch except bugfixes and hotfixes (see below). This is the branch from which the official releases are created, for instance version 4.1.0, 4.1.1, 4.1.2, etc.
- **the development branch, eg: '4.2.x_dev'** Note that the branch name differs from the stable branch by one increment on the minor revision number (2 instead of 1), and the '_dev' suffix is added for clarity. This is the development branch. It is *ahead* of the main branch, in the sense that it contains more recent commits than the main branch. It is thus the future state of the code. This is where the last developments of the code of PyMoDAQ are pushed. When the developers are happy with the state of this branch, typically when they finished to develop a new functionality and they tested it, this will lead to a new *release* of PyMoDAQ (4.1.x -> 4.2.0 in our example). In practice, the branch will simply be renamed from 4.2.x_dev to 4.2.x, and a new branch 4.3.x_dev will be created to continue the cycle.

Temporary branches:

- **Feature, eg: 'feature/new_colors'**: Any additional feature should be done on a feature branch. They are created based on the current development branch. When the feature is complete, a Pull Request must be open to integrate the changes into the development branch.
- **Bugfix, eg: 'bugfix/remove_annoying_message'**: These branches are meant to correct small issues. It can be created based on either the stable or development branch, depending on where the bug is located. Regardless, any bugfix must then be applied to all branches, if applicable (see note below).
- **Hotfix, eg: 'hotfix/fix_huge_bug'**: This is similar to a bugfix, but for more important bugs. More precisely, hotfixes are important enough that when applied, they will trigger an immediate new release (e.g. 4.1.1 -> 4.1.2) that incorporate the fix. At the contrary bugfixes can wait for a future release.

Note: Applying fixes across several branches

Let's consider the case where a bug is found on the **stable** branch. We create a new branch to fix it, open a pull request into the stable branch, and wait for it to be accepted. However, it is likely that the buggy code is also part of the **development** version, requiring another pull request on that branch! Thus, but when a bug is found, one should always remember to check if it is present on several branches.

Where to contribute

There are easy places where to contribute and some more obscure places... After a few years of code rewriting/enhancing, several places are available for easily adding functionalities. These places are implementing one form or another of the [Factory Pattern](#). For other places, you'll have to read the API documentation :-)

Factory Patterns (to be completed)

Data Exporting

New Exporting data format from the H5Browser is made easy see [pymodag/utis/h5modules/exporters](#)

Math functions in ROI

Scanning modes

Contributors

Here is a list of the main contributors:

Main modules

Functionalities

- Sébastien Weber, Research Engineer at CEMES/CNRS
- David Breteau, Research Engineer at Attolab facility, CEA Saclay
- Nicolas Tappy, Engineer at Attolight (<https://attolight.com/>)

Cleaning

- Sébastien Weber, Research Engineer at CEMES/CNRS
- David Trémouilles, Researcher at LAAS/CNRS

Plugins

- Sébastien Weber, Research Engineer at CEMES/CNRS
- Sophie Meuret, Researcher at CEMES/CNRS
- David Breteau, Research Engineer at Attolab facility, CEA Saclay
- and many others...

Extensions

- Sébastien Weber, Research Engineer at CEMES/CNRS
- Romain Geneaux, Researcher at CEA Saclay contributed to the PyMoDAQ-Femto extension

Documentation

- Sébastien Weber, Research Engineer at CEMES/CNRS
- Matthieu Cabos helped with this documentation
- David Breteau wrote the documentation of the PID extension and the tutorial: *Story of an instrument plugin development*

Testing

- Sébastien Weber, Research Engineer at CEMES/CNRS
- Pierre Jannot wrote tests with a total of 5000 lines of code tested during his internship at CEMES in 2021

Note: If you're not in the list and contributed somehow, sorry for that and let us know at sebastien.weber@cemes.fr

7.4.2 Plugins

A *plugin* is a python package whose name is of the type: `pymodaq_plugins_apluginname` containing functionalities to be added to PyMoDAQ

Note: A plugin may contains added functionalities such as:

- **Classes to add a given instrument:** allows a given instrument to be added programmatically in a *Control Modules* graphical interface
- **Instrument drivers** located in a *hardware* folder: contains scripts/classes to ease communication with the instrument. Could be third party packages such as Pymeasure
- **PID models** located in a *models* folder: scripts and classes defining the behaviour of a given PID loop including several actuators or detectors, see *The PID Model*
- **Extensions** located in a *extensions* folder: scripts and classes allowing to build extensions on top of the *Dash-Board*

Entry points python mechanism is used to let know PyMoDAQ of installed Instrument, PID models or extensions plugins

Plugins package configuration file

See *Plugins configuration for default values*.

Instrument Plugins

Any new hardware has to be included in PyMoDAQ within a *plugin*. A PyMoDAQ's plugin is a python package containing several added functionalities such as instruments objects. A instrument object is a class inheriting from either a `DAQ_Move_Base` or a `DAQ_Viewer_Base` class and implementing mandatory methods for easy and quick inclusion of the instrument within the PyMoDAQ control modules.

Plugins are articulated given their type: Moves or Viewers and for the latter their main dimensionality: **0D**, **1D** or **2D**. It is recommended to start from the *template repository* that includes templates for all kind of instruments and also the generic structure to build and publish a given plugin.

You will find below some information on the **how to** but comparison with existing plugins packages will be beneficial.

Note: You'll find in this documentation a detailed tutorial on *Story of an instrument plugin development*.

Installation

The main and official list of plugins is located in the [pymodaq_plugin_manager](#) repository on github. This constitutes a list of (contributed) python package that can be installed using the *Plugin Manager* (or directly using pip). Other unofficial plugins may also be installed if they follow PyMoDAQ's plugin specifications but you are invited to let know other users of the plugins you develop in order to contribute to PyMoDAQ's development.

PyMoDAQ is looking at startup for all installed packages that it can consider as its plugins. This includes by default the *pymodaq_plugins_mock* package of mock instruments installed on the *site_packages* location in your python distribution.

Contributions

If you wish to develop a plugin specific to a new hardware or feature not present on the github repo (and I strongly encourage you to do so!!), you will have to follow the rules as below.

Two cases are possible: either you want to add a new hardware from a manufacturer for which a repository already exists 1) (thorlabs, PI, Andor...) or not 2)

1. You have to fork the existing repo
2. you will use the [pymodaq_plugins_template](#) on github to create a new repo (see also the *How to create a new plugin/package for PyMoDAQ?* tutorial)

Once you've done that, you can clone the package locally and install it in developer using `pip install -e .` from the command line where you `cd` within the cloned package. This command will install the package but any change you apply on the local folder will be applied on the package. Then just add a new python file in the correct location.

Once you're ready with a working plugin, you can then:

1. Publish your repo on pypi (just by doing a release on github will trigger the creation of a pypi repository)
2. do a pull request on the initial repository to merge your new implementations.

Note: Starting with PyMoDAQ version 4.1.0 onwards, old github actions for publication and suite testing should be updated in the plugin packages. You can just use the one from the template repository

All the packages published on pypi using the template and the naming convention will be available in the plugin manager.

A very detailed tutorial has been published in this documentation: *Story of an instrument plugin development* and you can in the mean time look at this [video](#)

Naming convention

For an instrument plugin to be properly recognised by PyMoDAQ, the location and name of the underlying script must follow some rules and syntax. The [plugin template package](#) could be copied locally as a starting point:

- The plugin package will be named `pymodaq_plugins_xxxx` (name: `xxxx`)
- An actuator plugin (name: `xxxx`) will be a script whose name is `daq_move_Xxxx` (notice first X letter is capital)
- The main instrument class within the script will be named `DAQ_Move_Xxxx` (notice the capital letters here as well and sorry if it is troublesome)
- A detector plugin of dimensionality N (N=0, 1, 2 or N) (name: `xxxx`) will be a script whose name is `daq_NDviewer_Xxxx` (notice first X letter is capital, and replace N by 0, 1, 2 or leave it for higher dimensionality)

- The main instrument class within the script will be named `DAQ_NDViewer_Xxxx` (notice the capital letters here as well)

Hardware Settings

An important feature similar for all modules is the layout as a tree structure of all the hardware parameters. These settings will appear on the UI as a tree of parameters with a title and different types, see Fig. 7.98. On the module side, they will be instantiated as a list of dictionaries and later exist in the object `self.settings`. This object inherits from the `Parameter` object defined in `pyqtgraph`.

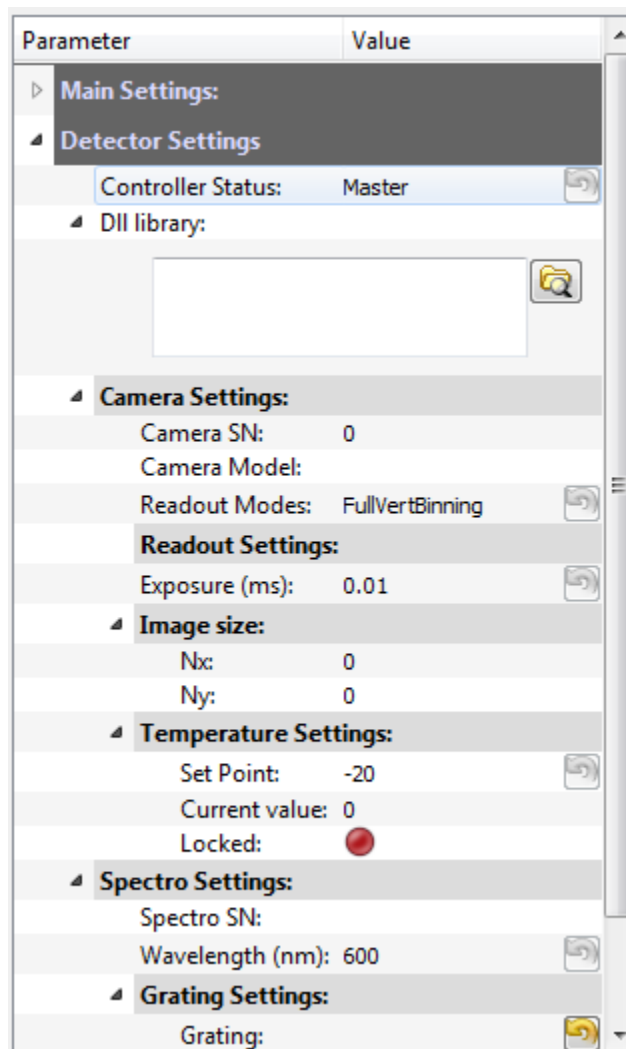


Fig. 7.98: Typical hardware settings represented as a tree structure (here from the `daq_2Dviewer_AndorCCD` plugin)

Here is an example of such a list of dictionaries corresponding to Fig. 7.98:

```
[{'title': 'Dll library:', 'name': 'andor_lib', 'type': 'browsepath', 'value': libpath},
 {'title': 'Camera Settings:', 'name': 'camera_settings', 'type': 'group', 'expanded': True, 'children': [
```

(continues on next page)

(continued from previous page)

```

    {'title': 'Camera SN:', 'name': 'camera_serialnumber', 'type': 'int', 'value': 0,
↪ 'readonly': True},
    {'title': 'Camera Model:', 'name': 'camera_model', 'type': 'str', 'value': '',
↪ 'readonly': True},
    {'title': 'Readout Modes:', 'name': 'readout', 'type': 'list', 'values': [
↪ 'FullVertBinning', 'Imaging'], 'value': 'FullVertBinning'},
    {'title': 'Readout Settings:', 'name': 'readout_settings', 'type': 'group',
↪ 'children': [
        {'title': 'single Track Settings:', 'name': 'st_settings', 'type': 'group',
↪ 'visible': False, 'children': [
            {'title': 'Center pixel:', 'name': 'st_center', 'type': 'int', 'value': 1 ,
↪ 'default': 1, 'min': 1},
            {'title': 'Height:', 'name': 'st_height', 'type': 'int', 'value': 1 ,
↪ 'default': 1, 'min': 1},
        ]}], ]}]

```

The list of available types of parameters (defined in `pymodaq.utils.parameter.pymodaq_ptypes.py`) is:

- `group` : “camera settings” on Fig. 7.98 is of type group
- `int` : settable integer (SpinBox_Custom object)
- `float` : settable float (SpinBox_Custom object)
- `str` : a QLineEdit object (see Qt5 documentation)
- `list` : “Readout Modes” Fig. 7.98 is a combo box
- `bool` : checkable boolean
- `bool_push` : a checkable boolean in the form of a QPushButton
- `led` : non checkable boolean in the form of a green (True) of red (False) led
- `led_push` : checkable boolean in the form of a green (True) of red (False) led
- `date_time` : a QDateTime object (see Qt5 documentation)
- `date` : a QDate object (see Qt5 documentation)
- `time` : a QTime object (see Qt5 documentation)
- `slide` : a combination of a slide and spinbox for floating point values (linear of log scale)
- `itemselect` : an object to easily select one or more items among a few
- `browsepath` : a text area and a pushbutton to select a given path or file
- `text` : a text area (for comments for instance)

Important: the *name* key in the dictionaries must **not** contain any space, please use underscore if necessary!

Note: For a live example of these Parameters and their widget, type in `parameter_example` in your shell or check the example folder

Once the module is initialized, any modification on the UI hardware settings will be send to the plugin through the `commit_settings` method of the plugin class and illustrated below (still from the `daq_2Dviewer_AndorCCD` plugin). The `param` method argument is of the type `Parameter` (from `pyqtgraph`):

```
def commit_settings(self,param):
    """
    | Activate parameters changes on the hardware from parameter's name.
    """
    try:
        if param.name()=='set_point':
            self.controller.SetTemperature(param.value())

        elif param.name() == 'readout' or param.name() in custom_parameter_tree.iter_
↪ children(self.settings.child('camera_settings', 'readout_settings')):
            self.update_read_mode()

        elif param.name()=='exposure':
            self.controller.SetExposureTime(self.settings.child('camera_settings',
↪ 'exposure').value()/1000) #temp should be in s
            (err, timings) = self.controller.GetAcquisitionTimings()
            self.settings.child('camera_settings','exposure').setValue(timings['exposure
↪ ']*1000)
        elif param.name() == 'grating':
            index_grating = self.grating_list.index(param.value())
            self.get_set_grating(index_grating)
            self.emit_status(ThreadCommand('show_splash', ["Setting wavelength"]))
            err = self.controller.SetWavelengthSR(0, self.settings.child('spectro_
↪ settings','spectro_wl').value())
            self.emit_status(ThreadCommand('close_splash'))
```

Emission of data

When data are ready (see [Data ready?](#) to know about that), the plugin has to notify the viewer module in order to display data and eventually save them. For this PyMoDAQ use two types of signals (see pyqt signal documentation for details):

- `dte_signal_temp`
- `dte_signal`

where dte stands for DataToExport, see [DataToExport](#).

Note: So far (07/07/2023) instrument plugins would use signals below to emit a list of DataFromPlugins objects

- `data_grabed_signal_temp` (old style, will be deprecated)
- `data_grabed_signal` (old style, will be deprecated)

It will be deprecated in versions > 4.1, as the object to use and emit are now DataToExport objects

They both *emit* the same type of signal but will trigger different behaviour from the viewer module. The first is to be used to send temporary data to update the plotting but without triggering anything else (so that the DAQ_Scan still awaits for data completion before moving on). It is also used in the initialisation of the plugin in order to preset the type and number of data viewers displayed by the viewer module. The second signal is to be used once data are fully ready to be send back to the user interface and further processed by DAQ_Scan or DAQ_Viewer instances. The code below is an example of emission of data:

```

from pymodaq.utils.data import Axis, DataFromPlugins, DataToExport
x_axis = Axis(label='Wavelength', units= "nm", data = vector_X)
y_axis = Axis(data=vector_Y)
self.dte_signal.emit(DataToExport('mydata', data=[
    DataFromPlugins(name='Camera',data=[data2D_0, data2D_1,...],
                    dim='Data2D', x_axis=x_axis,y_axis=y_axis),
    DataFromPlugins(name='Spectrum',data=[data1D_0, data1D_1,...],
                    dim='Data1D', x_axis=x_axis, labels=['label0', 'label1', ...]),
    DataFromPlugins(name='Current',data=[data0D_0, data0D_1,...],
                    dim='Data0D'),
    DataFromPlugins(name='Datacube',data=[dataND_0, dataND_1,...],
                    dim='DataND', nav_indexes=(0,2),
                    axes=[Axis(data=.., label='Xaxis', units= "µm", index=0)]))

```

Such an emitted signal would trigger the initialization of 4 data viewers in the viewer module. One for each `DataFromPlugins` in the data attribute (which is a list of `DataFromPlugins`). The type of data viewer will be determined by the `dim` key value while its name will be set to the `name` parameter value, for more details on data objects, see [What is PyMoDAQ's Data?](#)

Note: *New in version 4.1.0*

Deprecated in version 4.2.0, but still working

The behaviour of the `DAQ_Viewer` can be even more tailored using two extra boolean attributes in the `DataWithAxes` objects.

- `save`: will tell the `DAQ_Viewer` whether it should save the corresponding dwa (short for `DataWithAwes`)
- `plot`: will tell the `DAQ_Viewer` whether it should plot the corresponding dwa

New in version 4.2.0

the `save` and `plot` extra-attributes have been replaced by:

- `do_save`: will tell the `DAQ_Viewer` whether it should save the corresponding dwa (short for `DataWithAwes`)
- `do_plot`: will tell the `DAQ_Viewer` whether it should plot the corresponding dwa

`DataFromPlugins` objects have these two extra attributes by default with values set to `True`

Data ready?

One difficulty with these viewer plugins is to determine when data is ready to be read from the controller and then to be send to the user interface for plotting and saving. There are a few solutions:

- **synchronous**: The simplest one. When the `grab` command has been send to the controller (let's say to its `grab_sync` method), the `grab_sync` method will hold and freeze the plugin until the data are ready. The Mock plugin work like this.
- **asynchronous**: There are 2 ways of doing asynchronous *waiting*. The first is to poll the controller state to check if data are ready within a loop. This polling could be done with a while loop but if nothing more is done then the plugin will still be freezed, except if one process periodically the Qt queue event using `QtWidgets.QApplication.processEvents()` method. The polling can also be done with a timer event, firing periodically a check of the data state (ready or not). Finally, the nicest/hardest solution is to use callbacks (if the controller provides one) and link it to a `emit_data` method.

Synchronous example:

The code below illustrates the poll method using a loop:

```
def poll_data(self):
    """
    Poll the current data state
    """
    sleep_ms=50
    ind=0
    data_ready = False
    while not self.controller.is_ready():
        QThread.msleep(sleep_ms)

        ind+=1

        if ind*sleep_ms>=self.settings.child('timeout').value():

            self.emit_status(ThreadCommand('raise_timeout'))
            break

    QtWidgets.QApplication.processEvents()
    self.emit_data()
```

Asynchronous example:

The code below is derived from *daq_Andor_SDK2* (in *andor* hardware folder) and shows how to create a thread waiting for data ready and triggering the emission of data

```
class DAQ_AndorSDK2(DAQ_Viewer_base):

    callback_signal = QtCore.Signal() #used to talk with the callback object
    ...

    def ini_camera(self):
        ...
        callback = AndorCallback(self.controller.WaitForAcquisition) # the callback is
        ↪ linked to the controller WaitForAcquisition method
        self.callback_thread = QtCore.QThread() #creation of a Qt5 thread
        callback.moveToThread(self.callback_thread) #callback object will live within
        ↪ this thread
        callback.data_sig.connect(self.emit_data) # when the wait for acquisition
        ↪ returns (with data taken), emit_data will be fired

        self.callback_signal.connect(callback.wait_for_acquisition) #
        self.callback_thread.callback = callback
        self.callback_thread.start()

    def grab(self, Naverage=1, **kwargs):
        ...
        self.callback_signal.emit() #trigger the wait_for_acquisition method
```

(continues on next page)

(continued from previous page)

```

def emit_data(self):
    """
    Function used to emit data obtained by callback.
    """
    ...
    self.dte_signal.emit(
        DataToExport('mydata',
                     data=[DataFromPlugins('Camera',
                                             data=[np.squeeze(self.data.reshape((sizey,
↪sizey)).astype(np.float))]]))

class AndorCallback(QQtCore.QObject):

    data_sig=QtCore.Signal()
    def __init__(self,wait_fn):
        super(AndorCallback, self).__init__()
        self.wait_fn = wait_fn

    def wait_for_acquisition(self):
        err = self.wait_fn()

        if err != 'DRV_NO_NEW_DATA': #will be returned if the main thread called
↪CancelWait
            self.data_sig.emit()

```

Documentation from Andor SDK concerning the WaitForAcquisition method of the dll:

unsigned int WINAPI WaitForAcquisition(void)

WaitForAcquisition can be called after an acquisition is started using StartAcquisition to put the calling thread to sleep until an Acquisition Event occurs.

It will use less processor resources than continuously polling with the GetStatus function. If you wish to restart the calling thread without waiting for an Acquisition event, call the function CancelWait.

Hardware averaging

By default, if averaging of data is needed the Viewer module will take care of it software wise. However, if the hardware controller provides an efficient method to do it (that will save time) then you should set the class field hardware_averaging to True.

```

class DAQ_NDViewer_Template(DAQ_Viewer_base):
    """
    Template to be used in order to write your own viewer modules
    """
    hardware_averaging = True #will use the accumulate acquisition mode if averaging
    #is True else averaging is done software wise

```

Live Mode

By default, the live *Grab* mode is done software wise in the core code of the DAQ_Viewer. However, if one want to send data as fast as possible, the live mode is possible within a plugin.

For this, the plugin class attribute, `live_mode_available`, should be set to `True`.

```
class DAQ_2DViewer_MockCamera(DAQ_Viewer_base):  
  
    live_mode_available = True
```

The method `grab_data` will then receive a named boolean parameter (in kwargs) called *live* that tells if one should grab or snap data. The MockCamera plugin illustrates this feature:

```
def grab_data(self, Naverage=1, **kwargs):  
    """Start a grab from the detector  
  
    Parameters  
    -----  
    Naverage: int  
        Number of hardware averaging (if hardware averaging is possible, self.hardware_  
→averaging should be set to  
        True in class preamble and you should code this implementation)  
    kwargs: dict  
        others optionals arguments  
    """  
    if 'live' in kwargs:  
        if kwargs['live']:  
            self.live = True  
            # self.live = False # don't want to use that for the moment  
  
    if self.live:  
        while self.live:  
            data = self.average_data(Naverage)  
            QThread.sleep(kwargs.get('wait_time', 100))  
            self.dte_signal.emit(data)  
            QtWidgets.QApplication.processEvents()
```

Hardware needed files

If you are using/referring to custom python wrappers/dlls... within your plugin and need a place where to copy them in PyMoDAQ, then use the `\hardware` folder of your plugin package. For instance, the `daq_2Dviewer_AndorCCD` plugin need various files stored in the `andor` folder (on github repository). I would therefore copy it as `\pymodaq_plugins_andor\hardware\andor` and call whatever module I need within (meaning there is a `__init__.py` file in the *andor* folder) as:

```
#import controller wrapper  
from pymodaq_plugins.hardware.andor import daq_AndorSDK2 #this import the module DAQ_  
→AndorSDK2 containing classes, methods...  
#and then use it as you see fit in your module
```


Actuator plugin having multiple axis

See also: *Multiaxes controller*

When an actuator's controller can drive multiple axis (like a XY translation stage for instance), the plugin instrument class should defines two class attributes:

- `is_multiaxis` should be set to `True`. This will trigger the display of the multiaxis section on the UI
- `axes_names` should be a list or dict describing the different actuator such a controller can drive

```
class DAQ_Move_MockNamedAxes(DAQ_Move_base):
    is_multiaxes = True
    _axis_names = ['Xaxis', 'Yaxis', 'Zaxis']
    # or:
    _axis_names = {'Xaxis': 0, 'Yaxis': 1, 'Zaxis': 2}
```

would produce such display on the UI (Fig. Fig. 7.99):

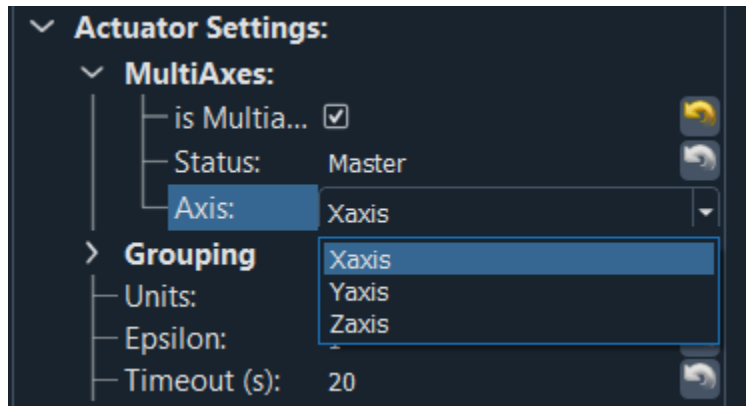


Fig. 7.99: Typical multiaxis settings represented as a combo box

Both the list or the dictionary will produce the same output on the UI but their use will depend of the controller and underlying methods of its driver to act on a particular axis. In the drivers derived from C code, methods will have an argument describing a particular axis as an integer. It is however not possible to pass integers directly to the combobox of the UI who holds strings. To deal with that *pyqtgraph*, and therefore *pymodaq*, uses a dictionary mapping the names of the axis (to be printed in the UI) to objects (here integers) to be used with the drivers's method.

A set of methods/properties have been introduced to quickly manipulate those and get either the current axis name of associated *value*.

Case of a list of strings:

```
>>> self.axis_name
'Yaxis'
>>> self.axis_names
['Xaxis', 'Yaxis', 'Zaxis']
>>> self.axis_value
'Yaxis'
```

Case of a dictionary of strings/integers:

```
>>> self.axis_name
'Yaxis'
>>> self.axei_names
{'Xaxis': 0, 'Yaxis': 1, 'Zaxis': 2}
>>> self.axis_value
1
```

Modifying the UI from the instrument plugin class

The user interface control module and the instrument plugin class are not in the same thread, moreover, the plugin class is not aware of the UI object (DAQ_Move or DAQ_Viewer). The following shows, how the instrument plugin class DAQ_Move_MyPluginClass relates to DAQ_Move:

```
class DAQ_Move:
    def init_hardware(self...):
        hardware: DAQ_Move_Hardware
        hardware.moveToThread()

class DAQ_Move_Hardware:
    hardware: DAQ_Move_Base

class DAQ_Move_MyPluginClass(DAQ_Move_Base):
    """Plugin class defined in a plugins repository."""
```

Therefore, you'll find below ways to interact with the UI from the plugin class.

The most generic way (valid for both control modules) is to use the `emit_status` method, defined in the parent class of the instrument plugin class. Such a method takes one argument, a `ThreadCommand` and will send this object to the `thread_status` method of the UI main class.

Note: A *ThreadCommand* is an object taking two arguments a string (the command) and a named attribute called attribute that can be any type. This *ThreadCommand* is used everywhere in PyMoDAQ to communicate between threads.

Control modules share some commands, see `thread_status`

- **Update_status:** call the `update_status` method with status attribute as a string
- **close:** close the current thread and delete corresponding attribute on cascade.
- **update_main_settings:** update the main settings in the UI settings tree
- **update_settings:** update the actuator's settings in the UI settings tree
- **raise_timeout:** call the `raise_timeout` method
- **show_splash:** show the splash screen displaying info from the argument attributes of the command
- **close_splash:** close the splash screen

Splash Screen and info

You can therefore show info about initialization in a splash screen using (taken from the Mock ODViewer plugin):

```
self.emit_status(ThreadCommand('show_splash', 'Starting initialization'))
QtCore.QThread.msleep(500)
self.ini_detector_init(old_controller=controller,
                       new_controller='Mock controller')
self.emit_status(ThreadCommand('show_splash', 'generating Mock Data'))
QtCore.QThread.msleep(500)
self.set_Mock_data()
self.emit_status(ThreadCommand('update_main_settings', [['wait_time'],
                                                         self.settings.child('wait_time').
                                                         ↪value(), 'value'])))
self.emit_status(ThreadCommand('show_splash', 'Displaying initial data'))
QtCore.QThread.msleep(500)
# initialize viewers with the future type of data
self.dte_signal_temp.emit(DataToExport('Mock0D', data=[DataFromPlugins(name='Mock1', ↪
                                                         ↪data=np.array([0])),
                                                         dim='Data0D',
                                                         labels=['Mock1',
                                                         ↪'label2']]])))
self.emit_status(ThreadCommand('close_splash'))
```

Modifying the UI settings

if you want to modify the settings tree of the UI (the *Main Settings* part as the other one, you can do so within the plugin directly), you can do so using:

```
self.emit_status(ThreadCommand('update_main_settings', [['wait_time'], 10, 'value'])))
```

The attribute of the ThreadCommand is a bit complex here [['wait_time'], 10, 'value']. It is a list of three variables:

- a list of string defining a path in the main_settings tree hierarchy
- an object (here an integer)
- a string specifying the type of modification, either:
 - value: the object should therefore be the new value of the modified parameter
 - limits: the object should be a sequence listing the limits of the parameter (depends on the type of parameter)
 - options: the object is a dictionary defining the options to modify
 - childAdded: the object is a dictionary generated using SaveState of a given Parameter

DAQ_Move specific commands

Specifics commands for the DAQ_Move are listed in: `thread_status` and explained a bit below

- **ini_stage**: obtains info from the initialization
- **get_actuator_value**: update the UI current value
- **move_done**: update the UI current value and emits the `move_done` signal
- **outofbounds**: emits the `bounds_signal` signal with a `True` argument
- **set_allowed_values**: used to change the behaviour of the spinbox controlling absolute values, see `set_abs_spinbox_properties`
- **stop**: stop the motion

You can directly modify the printed current actuator's value using the `emit_value(12.4)` method which is a shortcut of `emit_status(ThreadCommand('get_actuator_value', 12.4))`. In that case the printed value would show 12.4.

You can also modify some SpinBox of the UI (the ones used to specify the absolute values) using the `set_allowed_values` command. In that case the attribute argument of the `ThreadCommand` should be a dictionary, see `set_abs_spinbox_properties`.

DAQ_Viewer specific commands

Specifics commands for the DAQ_Viewer are listed in: `thread_status` and explained a bit below

- **ini_detector**: update the status with “detector initialized” value and init state if attribute not null.
- **grab** : emit `grab_status(True)`
- **grab_stopped**: emit `grab_status(False)`
- **init_lcd**: display a LCD panel
- **lcd**: display on the LCD panel, the content of the attribute
- **stop**: stop the grab

The interesting bit is the possibility to display a *LCD widget* to display some numerical values (could be 0D Data also emitted using the `dte_signal` but could also be any values). You should first init the LCD screen using the command: `init_lcd` with an attribute being a dictionary with keys either:

- **digits**: an integer specifying the number of digits to display
- **Nvals**: the number of numerical values to be displayed
- **labels**: the name/label of each value

For instance, in the 0D Mock viewer plugin:

```
if not self.lcd_init:
    self.emit_status(ThreadCommand('init_lcd', dict(labels=['dat0', 'data1'], Nvals=2,
→ digits=6)))
    QtWidgets.QApplication.processEvents()
    self.lcd_init = True
self.emit_status(ThreadCommand('lcd', data_tot))
```

Where the `lcd` is first initialized, then data are sent using the `lcd` command taking as attribute a list of 0D numpy arrays

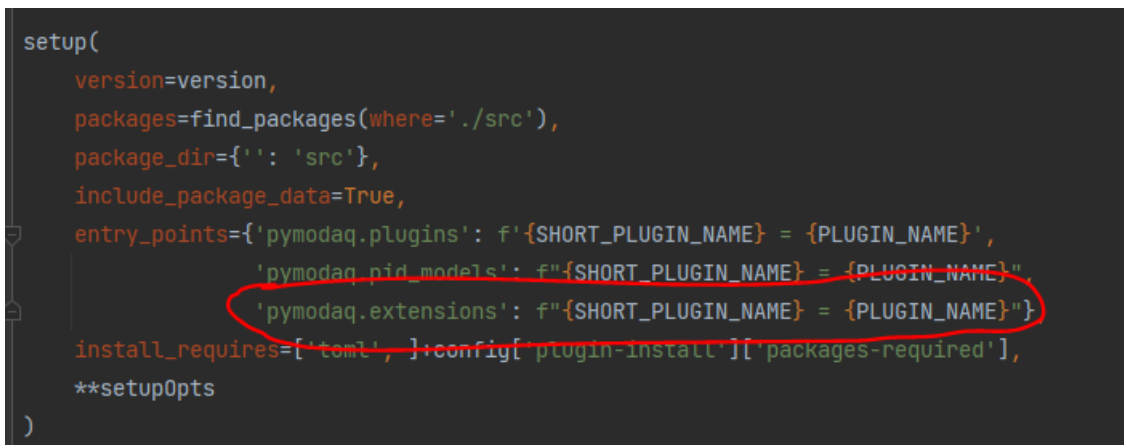
Extension Plugins

PyMoDAQ's plugins allows to add fonctionnalities to PyMoDAQ from external packages. You should be well aware of the instrument type plugins and somehow of the PID models plugins. Here we are highlighting how to built dashboard extensions such as the *DAQ Scan*.

For your package to be considered as a PyMoDAQ's dashboard extension, you should make sure of a few things:

- The entrypoint in the setup file should be correctly configured, see Fig. 7.100
- The presence of an *extensions* module at the root of the package
- each module within the *extensions* module will define an extension. It should contains three attributes:
 - EXTENSION_NAME: a string used to display the extension name in the dashboard extension menu
 - CLASS_NAME: a string giving the name of the extension class
 - a class deriving from the *CustomApp* base class (see *Custom App*)

The *pymodaq_plugins_template* contains already all this, so make sure to start from there when you wish to build an extension.



```

setup(
    version=version,
    packages=find_packages(where='./src'),
    package_dir={'': 'src'},
    include_package_data=True,
    entry_points={'pymodaq.plugins': f'{SHORT_PLUGIN_NAME} = {PLUGIN_NAME}',
                  'pymodaq.pid_models': f'{SHORT_PLUGIN_NAME} = {PLUGIN_NAME}',
                  'pymodaq.extensions': f'{SHORT_PLUGIN_NAME} = {PLUGIN_NAME}'},
    install_requires=[toml, ]+config['plugin-install']['packages-required'],
    **setupOpts
)
  
```

Fig. 7.100: The correct configuration of your package.

The class itself defining the extension derives from the *CustomApp* base class. As such, it's `__init__` method takes two attributes, a *DoackArea* instance and a *DashBoard* instance (the one from which the extension will be loaded and that contains all the actuators/detectors needed for your extension). The *DashBoard* will smoothly initialize your class when launching it from the menu. Below you'll find a sample of an extension module with an extension class called *MyExtension* (from the *pymodaq_plugins_template* package)

```

EXTENSION_NAME = 'MY_EXTENSION_NAME'
CLASS_NAME = 'MyExtension'

class MyExtension(gutils.CustomApp):
    # list of dicts enabling the settings tree on the user interface
    params = [
        {'title': 'Main settings:', 'name': 'main_settings', 'type': 'group', 'children
    ↪': [
        {'title': 'Save base path:', 'name': 'base_path', 'type': 'browsepath',
         'value': config['data_saving']['h5file']['save_path']},
  
```

(continues on next page)

(continued from previous page)

```

        {'title': 'File name:', 'name': 'target_filename', 'type': 'str', 'value': "
↪", 'readonly': True},
        {'title': 'Date:', 'name': 'date', 'type': 'date', 'value': QtCore.QDate.
↪currentDate()},
        {'title': 'Do something, such as showing data:', 'name': 'do_something',
↪'type': 'bool', 'value': False},
        {'title': 'Something done:', 'name': 'something_done', 'type': 'led', 'value
↪': False, 'readonly': True},
        {'title': 'Infos:', 'name': 'info', 'type': 'text', 'value': ""},
        {'title': 'push:', 'name': 'push', 'type': 'bool_push', 'value': False}
    ]],
    {'title': 'Other settings:', 'name': 'other_settings', 'type': 'group', 'children
↪': [
        {'title': 'List of stuffs:', 'name': 'list_stuff', 'type': 'list', 'value':
↪'first',
        'limits': ['first', 'second', 'third'], 'tip': 'choose a stuff from the list
↪'},
        {'title': 'List of integers:', 'name': 'list_int', 'type': 'list', 'value':
↪0,
        'limits': [0, 256, 512], 'tip': 'choose a stuff from this int list'},
        {'title': 'one integer:', 'name': 'an_integer', 'type': 'int', 'value': 500,
↪},
        {'title': 'one float:', 'name': 'a_float', 'type': 'float', 'value': 2.7, },
    ]],
]

def __init__(self, dockarea, dashboard):
    super().__init__(dockarea, dashboard)
    self.setup_ui()

```

7.4.3 Custom App

PyMoDAQ's set of modules is a very efficient way to build a completely custom application (related to data acquisition or actuators displacement) without having to do it from scratch. Fig. 7.101 is an example of such an interface build using only PyMoDAQ's building blocks. The corresponding script template is within the example folder.

Note: A generic base class *CustomApp* located in *pymodaq.utils.gui_utils* can be used to build very quickly standalone *Application* or *Dashboard* extensions. The *DAQ_Logger* extension has been built using it as well as some examples in the example folder.

Below you'll find the skeleton of a *CustomApp* subclassing the base class and methods you have to override with your App/Extension specifics:

```

class CustomAppExample(gutils.CustomApp):

    # list of dicts enabling a settings tree on the user interface
    params = [
        {'title': 'Main settings:', 'name': 'main_settings', 'type': 'group', 'children

```

(continues on next page)

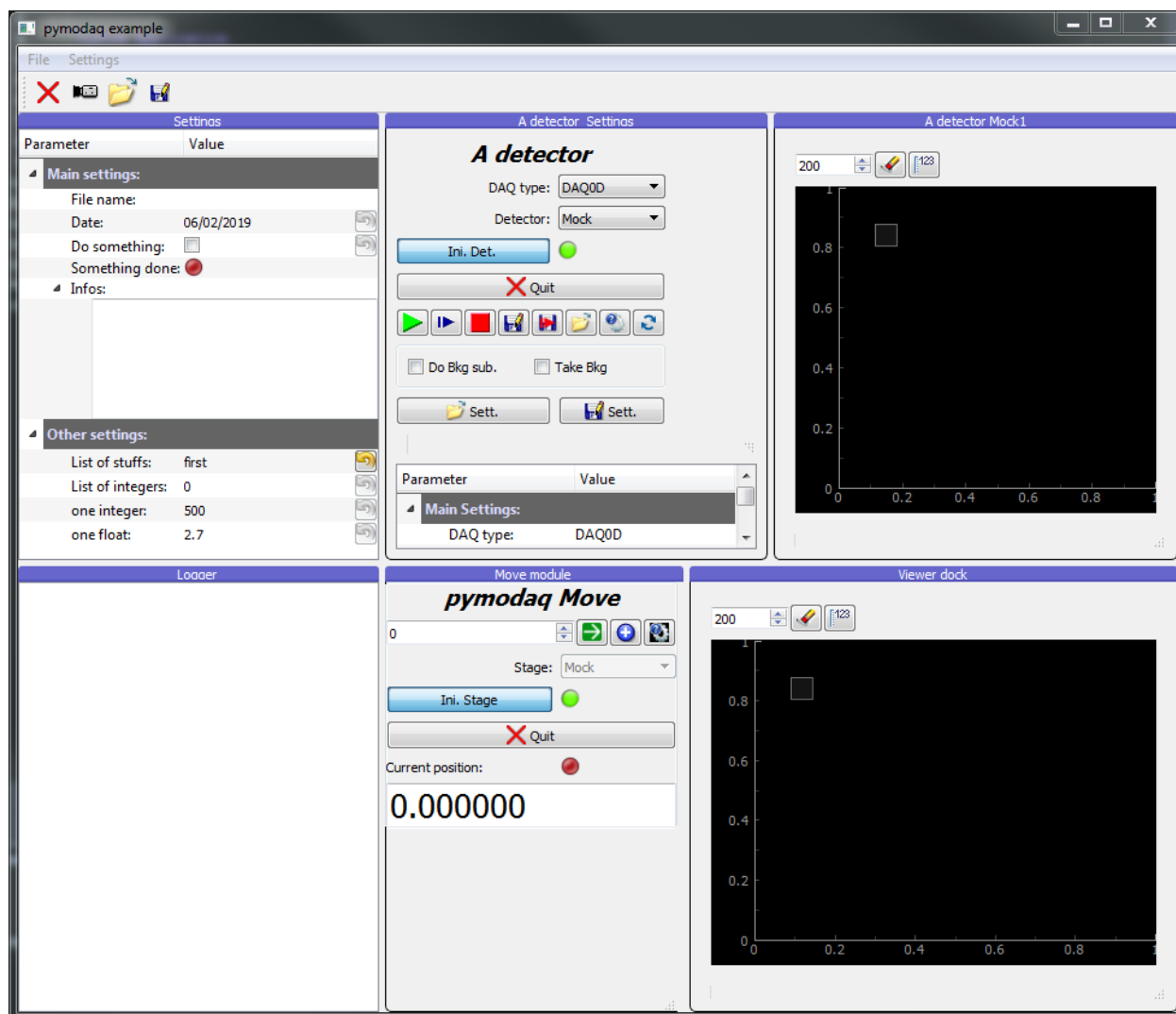


Fig. 7.101: A custom application build using PyMoDAQ's modules.

(continued from previous page)

```

↪': [
    {'title': 'Save base path:', 'name': 'base_path', 'type': 'browsepath',
     'value': config['data_saving']['h5file']['save_path']},
    {'title': 'File name:', 'name': 'target_filename', 'type': 'str', 'value': "
↪", 'readonly': True},
    {'title': 'Date:', 'name': 'date', 'type': 'date', 'value': QDate.
↪currentDate()},
    {'title': 'Do something, such as showing data:', 'name': 'do_something',
↪'type': 'bool', 'value': False},
    ]},
]

def __init__(self, dockarea, dashboard=None):
    super().__init__(dockarea)
    # init the App specific attributes
    self.raw_data = []

def setup_actions(self):
    """
    subclass method from ActionManager
    """
    logger.debug('setting actions')
    self.add_action('quit', 'Quit', 'close2', "Quit program", toolbar=self.toolbar)
    self.add_action('grab', 'Grab', 'camera', "Grab from camera", checkable=True,
↪toolbar=self.toolbar)
    logger.debug('actions set')

def setup_docks(self):
    """
    subclass method from CustomApp
    """
    logger.debug('setting docks')
    self.dock_settings = gutils.Dock('Settings', size=(350, 350))
    self.dockarea.addDock(self.dock_settings, 'left')
    self.dock_settings.addWidget(self.settings_tree, 10)
    logger.debug('docks are set')

def connect_things(self):
    """
    subclass method from CustomApp
    """
    logger.debug('connecting things')
    self.actions['quit'].connect(self.quit_function)
    self.actions['grab'].connect(self.detector.grab)
    logger.debug('connecting done')

def setup_menu(self):
    """
    subclass method from CustomApp
    """
    logger.debug('settings menu')
    file_menu = self.mainwindow.menuBar().addMenu('File')

```

(continues on next page)

(continued from previous page)

```

self.affect_to('quit', file_menu)
file_menu.addSeparator()
logger.debug('menu set')

def value_changed(self, param):
    logger.debug(f'calling value_changed with param {param.name()}')
    if param.name() == 'do_something':
        if param.value():
            self.settings.child('main_settings', 'something_done').setValue(True)
        else:
            self.settings.child('main_settings', 'something_done').setValue(False)

    logger.debug(f'Value change applied')

"""
All other methods required by your Application class
"""

```

In a few lines of codes, you'll get an application running. For the available *Parameter* available for your *settings_tree*, see *Settings*.

7.4.4 Managers and Mixin Objects

Parameter Manager

Action Manager

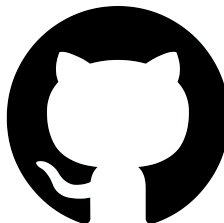
Modules Manager

ROI Manager

7.5 Tutorials

7.5.1 Git/GitHub

Author email	david.breseau@cea.fr
First edition	december 2023
Difficulty	Easy



Create an account & raise an issue on GitHub

We present here how to create a free account on GitHub so that as a user of PyMoDAQ, we will know where to ask for help to the PyMoDAQ community or report a bug.

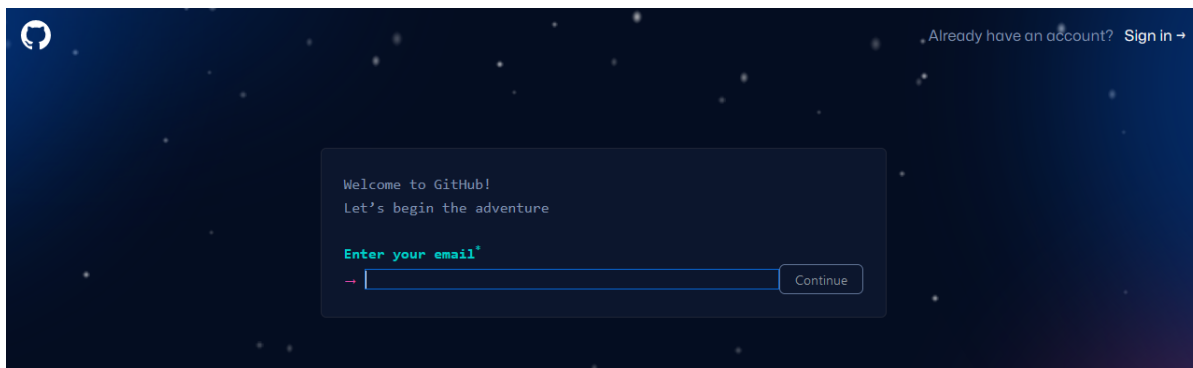
This tutorial does not require any knowledge of Python or GitHub.

What is GitHub?

[GitHub](#) is a free cloud service that **allows anyone to host an open-source project on distant servers**. PyMoDAQ is hosted on GitHub. As a PyMoDAQ user, we should know about it, even if we are not interested in the code. Indeed GitHub does not only host code, but it also proposes several features around it that allow to ease the communication within the community around the program (users or maintainers). So **if we face a problem with PyMoDAQ, that is where we should ask for help!**

Create an account

Creating an account on GitHub is necessary if we want to open a discussion or raise an issue. This is free of charge and does not commit us to anything. Let's go to the [GitHub website](#). On the top right of the page, let's click on *Sign up* and follow the guide.



Troubleshoot PyMoDAQ: raise an issue

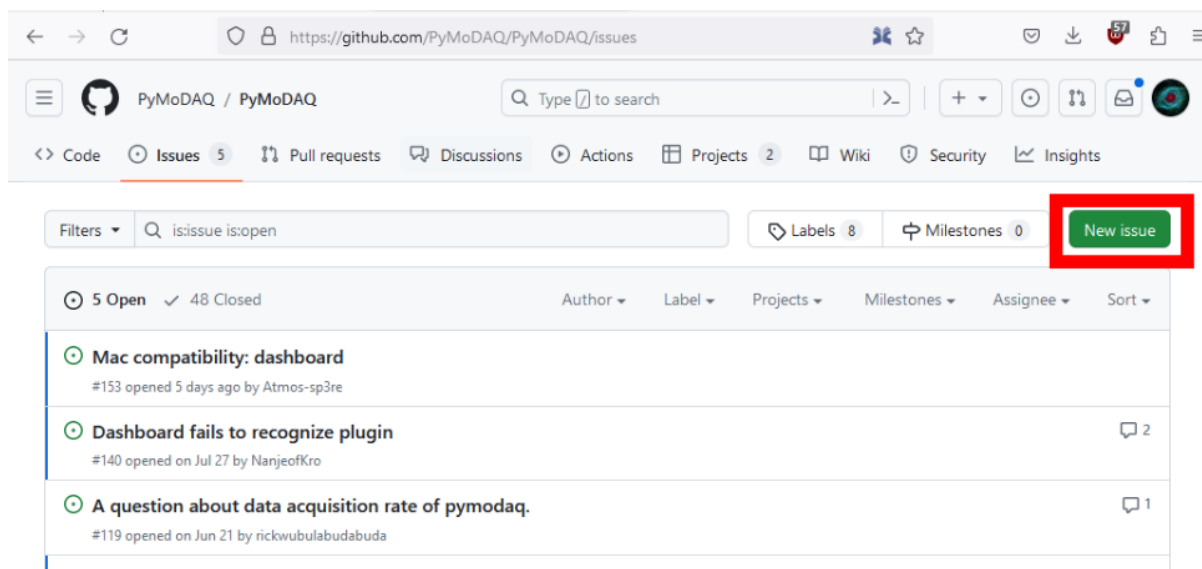
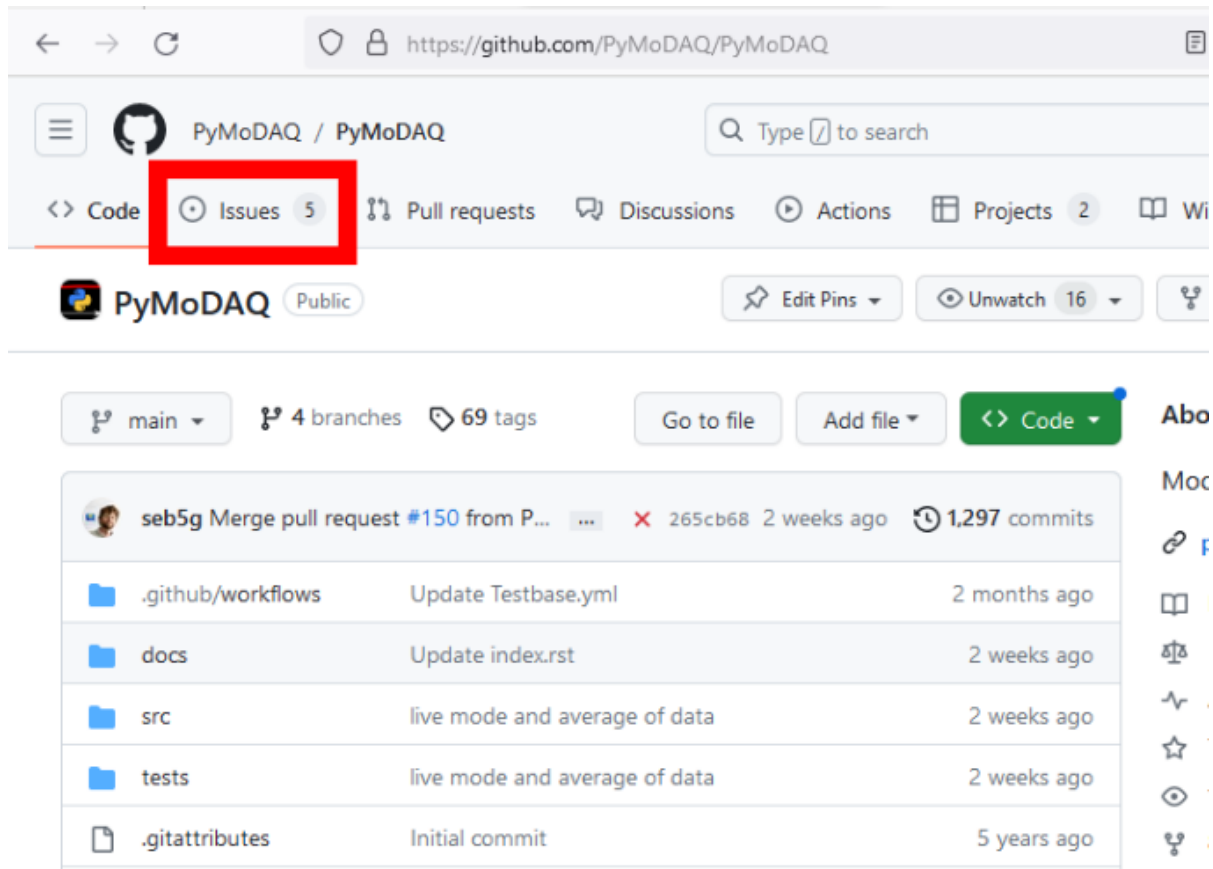
Now that we have our own account on GitHub and are logged in. Let's go to the [PyMoDAQ GitHub account](#).

Here we have access to all the code of PyMoDAQ, and all the history of its development. But what we are looking for now is the place where to ask for help in case we are in trouble. For this we should click on the *Issues* tab.

Anytime we face a problem or a bug in the program we can raise an issue. Describe as precisely as possible our problem. A discussion will be opened with the maintainers who will try to help us. This is the most efficient way to troubleshoot PyMoDAQ because the history of the issues is conserved, which could be helpful to solve future problems. This contributes to the documentation of the code. **We don't need to know the code to raise an issue, and it is really helpful to improve the stability of the program, so we should not hesitate to do so!**

Thanks to such functionalities, the PyMoDAQ GitHub account is the meeting point of the community around PyMoDAQ.

Author email	david.bresteau@cea.fr
Last update	january 2024
Difficulty	Easy





Basics of Git and GitHub

We introduce Git and GitHub in Pymodaq documentation because we believe that every experimental physicist should know about those wonderful tools that have been made by developers. They will help us code and share our code efficiently, not only within the framework of Pymodaq or even Python. Moreover, since Pymodaq is an open source project, its development is based on those tools. They have to be mastered if we want to contribute to the project or develop our own extension. Even as a simple user, we will learn where to ask for help when we are in difficulty, because Pymodaq's community is organized around those tools.

Why Git?

Git answers mainly two important questions:

How do I organize my code development efficiently? (local use)

- It allows you to come back to every version of your code.
- It forces you to document every step of the development of your code.
- You can try any modification of your code safely.
- It is an indispensable tool if you work on a bigger project than a few scripts.

How do I work with my colleagues on the same code? (remote use)

- Git tackles the general problem of several people working on the same project: it can be scientists working on a paper, some members of a parliament working on a law, some developers working on a program...
- It is a powerful tool that allows multiple developers to work on the same project without conflicting each other.
- It allows everyone that download an open-source project to have the complete history of its development.
- Coupled with a cloud-based version control service like GitHub, it allows to easily share your project with everyone, and have contributors, like PyMoDAQ!

How does it do that?

A program is nothing more than a set of files placed in the right subfolders.

Git is a *version control software*: it follows the development of a program (i.e. its different *versions*) by keeping track of every modifications of files in a folder.

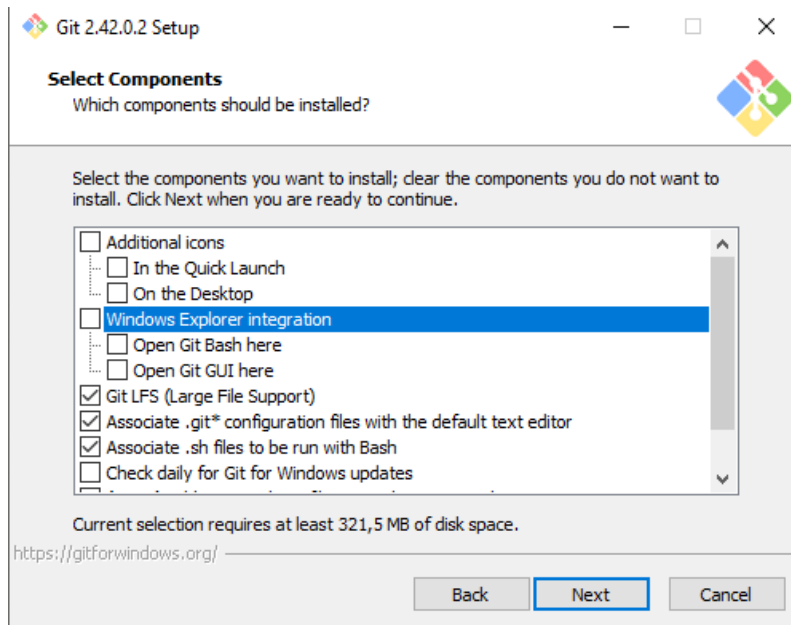
Installation & configuration for Windows

Installation

Fig. 7.102: Download the installer from the official website.

Download the installer from [the official website](https://gitforwindows.org/). Run the installer. From all the windows that will appear, let the default option, except for the following ones.

Uncheck “Windows Explorer integration”.



For the default editor, do not let Vim if you don't know about it, for example you can choose Notepad++.

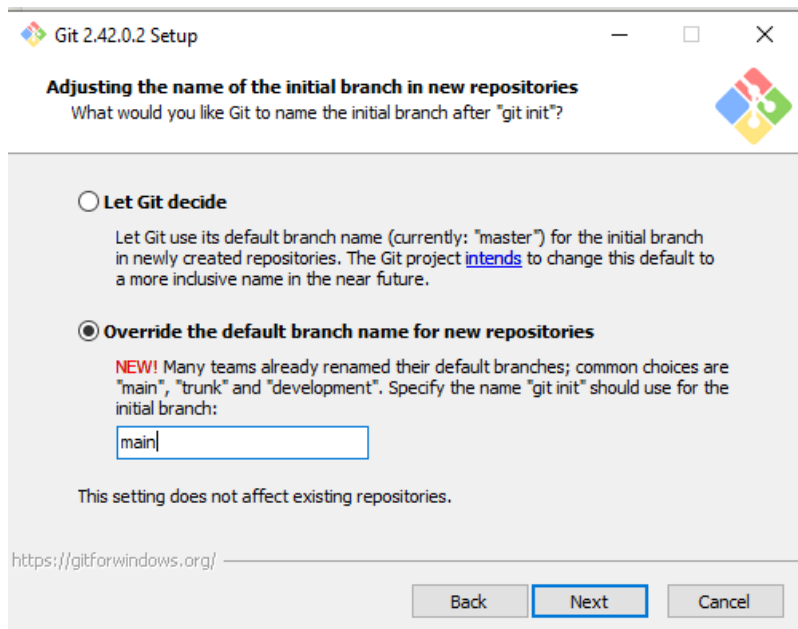
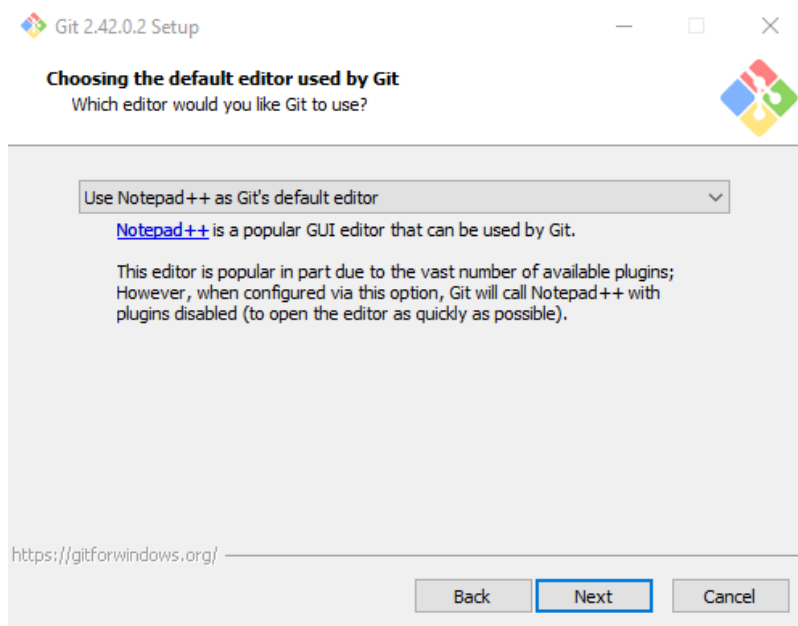
Use the following option for the name of the default branch.

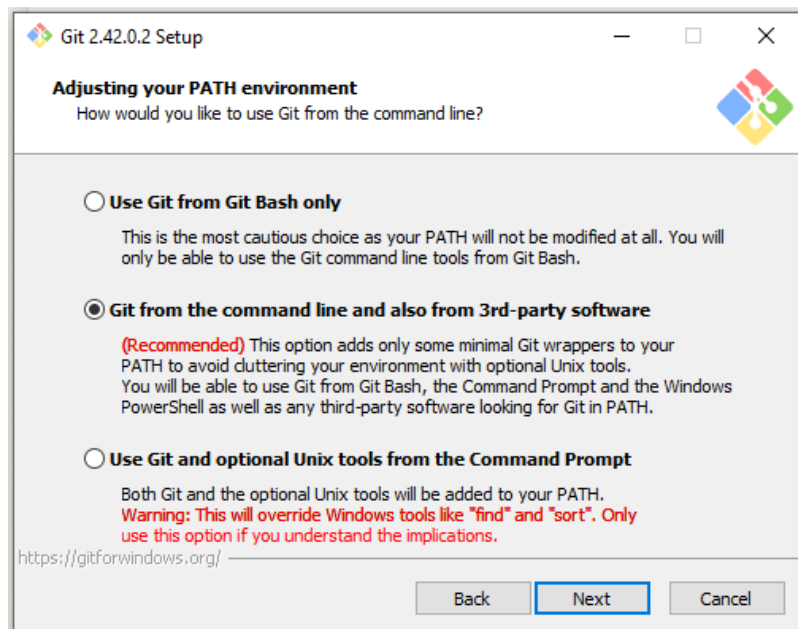
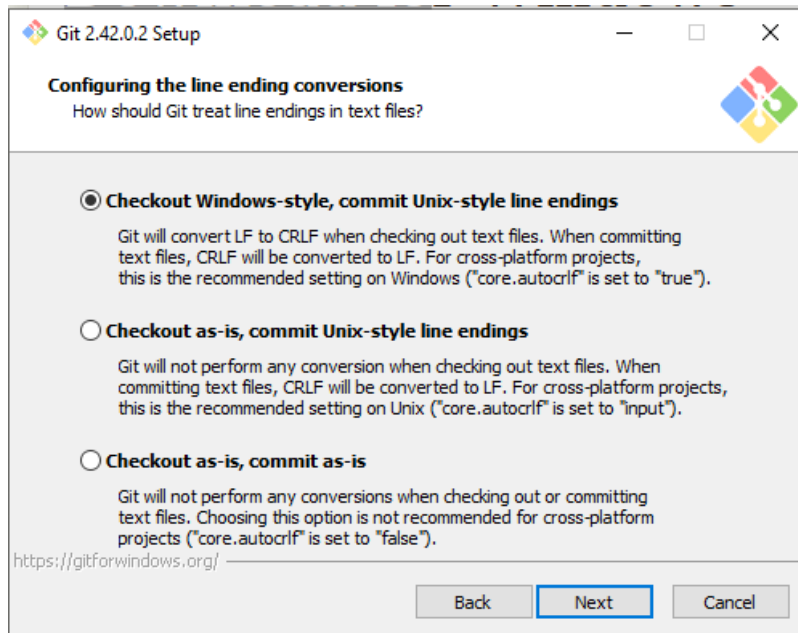
If you develop from Windows, it is better that you let Git manage the line endings.

Use the second option here.

Open the Git Bash terminal (Windows Applications > Git > Git Bash or Search for “Git Bash”) that has been installed with the Git installer.

We can now check that it is actually installed on our system.







The image shows a screenshot of a MINGW64 terminal window. The title bar indicates the path 'MINGW64:/c/Users/dbretea'. The terminal content shows the prompt 'dbretea@iram-fb-004873 MINGW64 ~' followed by a dollar sign '\$' and a vertical bar '|'. Below this, the command '\$ git --version' is entered, and the output 'git version 2.42.0.windows.2' is displayed.

Configuration

Just after the installation, you should configure Git so that he knows your email and name. This configuration is *global* in the sense that it does not depend on the project (the repository) you are working on. Use the following commands replacing with your own email and a name of your choice:

```
$ git config --global user.email "david.breseau@cea.fr"
$ git config --global user.name "David Breseau"
```

Good, we are now ready to use Git!

Installation & configuration for Ubuntu

Installation

In a terminal

```
$ sudo apt install git
```

Configuration

Just after the installation, you should configure Git so that he knows your email and name. This configuration is *global* in the sense that it does not depend on the project (the repository) you are working on. Use the following commands replacing with your own email and a name of your choice:

```
$ git config --global user.email "david.bresteau@cea.fr"
```

```
$ git config --global user.name "David Bresteau"
```

Good, we are now ready to use Git!

Local use of Git

We will start by using Git just on our local machine.

Before we start...

What kind of files I CAN track with Git?

Opened file formats that use text language: any “normal” language like C++, Python, Latex, markdown...

What kind of files I CANNOT track with Git?

- Closed file format like Word, pdf, Labview...
- Images, drawings...

The *init* command: start a new project

We start a project by creating a folder in our *home* directory, with the *mkdir* Bash command (for “make directory”)

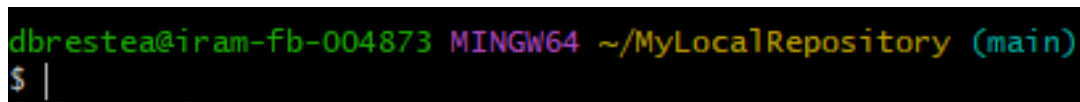
Note: The *home* directory corresponds to the directory that is reserved to the connected user. On Windows, it corresponds to the path *C:\Users\<username>*. Here the user is called *dbrestea*, you should replace it by your own username. When we open Git Bash, or any terminal in general, we are placed at our home directory in the file system, it can be represented by the *~* symbol (in orange in the above screenshots).

```
$ mkdir MyLocalRepository
```

And *cd* (for “change directory”) into this folder

```
$ cd MyLocalRepository
```

It should look like this now:



Now, we tell Git to track this folder with the *init* Git command

```
$ git init
```

Any folder that is tracked by Git contains a *.git* subfolder and called a *repository*.

We now create a new *my_first_amazing_file.txt* file in this folder and write *Hello world!* inside

The *status* command

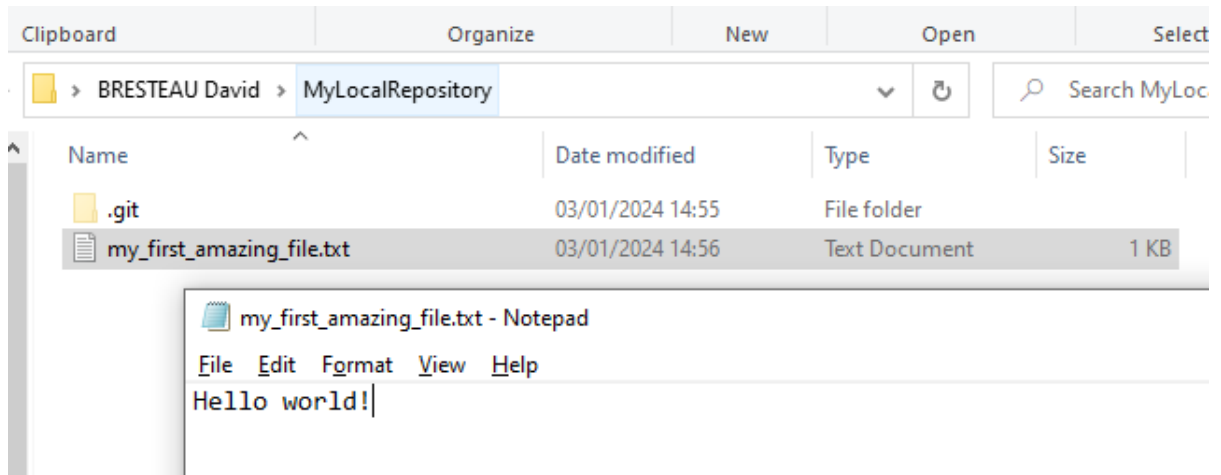
You should never hesitate to run this command, it gives you the current status of the project.

```
$ git status
```

It should look like this:

Here Git says that he noticed that we created a new file, but he placed it under the *Untracked files* and colored it in red.

The red means that Git does not know what to do with this file, he is waiting for an order from us.



```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    my_first_amazing_file.txt

nothing added to commit but untracked files present (use "git add" to track)
```

We have to tell him explicitly to track this file. To do so, we will just follow what he advised us, and use the *add* command.

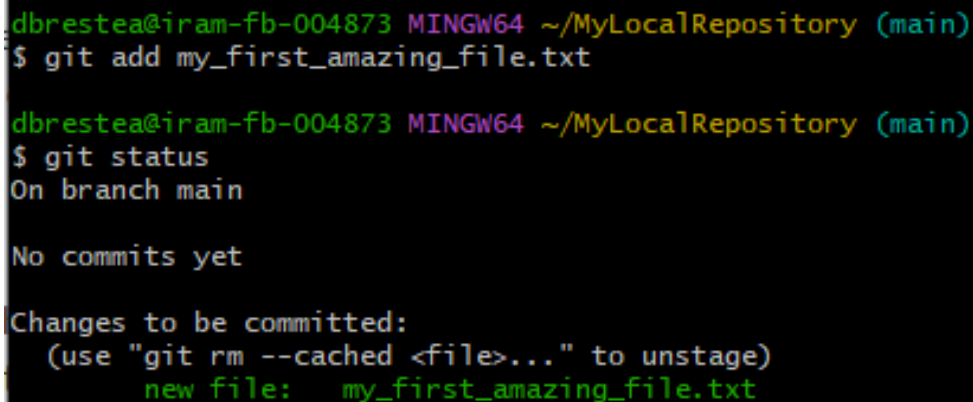
The *add* command

To put a file under the supervision of Git (to *track* the file), we use the *add* command. This has to be done only the first time we add a file into the folder.

```
$ git add my_first_amazing_file.txt
```

Then we do again the *status* command to see what have changed.

It should look like this:

A terminal window with a black background and green text. The prompt is 'dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)'. The first command is '\$ git add my_first_amazing_file.txt'. The second command is '\$ git status', which outputs 'On branch main', 'No commits yet', and 'Changes to be committed: (use "git rm --cached <file>..." to unstage) new file: my_first_amazing_file.txt'.

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git add my_first_amazing_file.txt

dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   my_first_amazing_file.txt
```

Now the filename turned green, which means that the file is tracked by Git and ready to be *committed*.

The *commit* command

A *commit* is a fundamental notion of Git.

A commit is a snapshot of the folder status at a point in time.

It is our responsibility to decide when to do a commit.

A commit should be done at every little change we do on our program, after we tested that the result is as we expected. For example, we should do a commit each time we add a new functionality to our program that is working properly.

For now, we just have one sentence in the file: “Hello world!”, but that’s a start. Let us do our initial commit with the following command

```
$ git commit -am "Initial commit of my amazing project. Add my first amazing file and say Hello world!"
```

It should look like this:

After the *-am* options (which means that we *add* the files (here we add the file *in the commit* and not in the tracking system of Git), and we type the *message* of our commit just after the command), we put a message to describe what we have done between parenthesis.

If we now look at the status of our project

Everything is clean. We just did our first commit! :)

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git commit -am "Initial commit of my amazing project. Add my first amazing file and say Hello world!"
[main (root-commit) 3378884] Initial commit of my amazing project. Add my first amazing file and say Hello world!
1 file changed, 1 insertion(+)
 create mode 100644 my_first_amazing_file.txt
```

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git status
On branch main
nothing to commit, working tree clean
```

The *log* command

The *log* command will give us the complete history of the commits since the beginning of the project.

```
$ git log
```

It should look like this:

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git log
commit 3378884d6a8994aeb07d10d8aa3c290f651572e6 (HEAD -> main)
Author: quantumm <david.bresteau@cea.fr>
Date:   Wed Jan 3 15:12:49 2024 +0100

    Initial commit of my amazing project. Add my first amazing file and say Hello world!
```

We can see that for each commit we have:

- An *id* that has been attributed to the commit, which is the big number in orange.
- The name and email address of the author.
- The date and time of the commit.
- The message that the author has written.

In the following we will use the *–oneline* option to get the useful information in a more compact way.

```
$ git log --oneline
```

It should look like this:

The *diff* command

The *diff* command is here to tell us what have changed since our last commit.

Let us now put some interesting content in our file. We will found this in the textart.me website. Let's choose an animal and copy paste it into our file. (Textart is the art of drawing something with some keyboard characters. It would be equivalent to just add a sentence in the file).

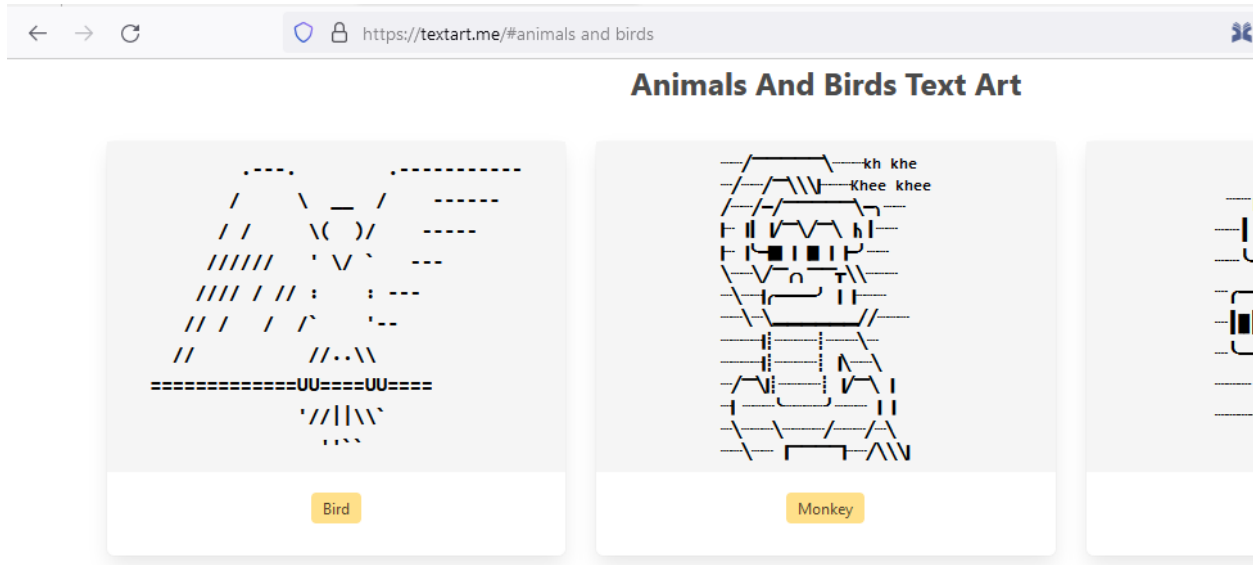
Let's go for the monkey, he is fun!

Let's not forget to save the file.

What happen if we ask for a difference from Git, with the *diff* command?

```
$ git diff
```

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git log --oneline
3378884 (HEAD -> main) Initial commit of my amazing project. Add my first amazing file and say Hello world!
```



It should look like this:

[illegible]

In *green* appears what we have added, in *red* appears what we have removed.

The *diff* command allows us to check what we have modified. Since we are happy with our last modification, we will commit our changes.

```
$ git commit -am "The funny monkey has been added."
```

Let us check what the log says now.

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git log --oneline
6045fb4 (HEAD -> main) The funny monkey has been added.
3378884 Initial commit of my amazing project. Add my first amazing file and say Hello world!
```

We now have two commits in our history.

The *revert* command

The *revert* command is here if we want to come back to a previous state of our folder.

Let's say that we are not happy with the monkey anymore. We would like to come back to the original state of the file just before we added the monkey. Since we did the things properly, by committing at every important point, this is a child play.

We use the *revert* command and the commit number that we want to cancel. The commit number is found by using the *log --oneline* command. In our case it is 6045fb4.

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git revert 6045fb4
[main ed7e909] Revert "The funny monkey has been added."
1 file changed, 1 insertion(+), 18 deletions(-)
```

This command will open Notepad++ (because we configured this editor in the installation section), just close it or modify the first text line if you want another commit message.



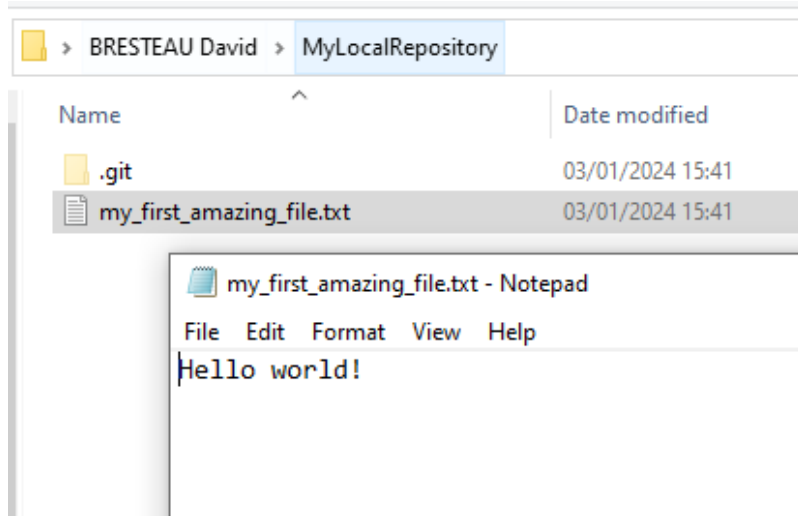
Let's now see the history

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git log --oneline
ed7e909 (HEAD -> main) Revert "The funny monkey has been added."
6045fb4 The funny monkey has been added.
3378884 Initial commit of my amazing project. Add my first amazing file and say Hello world!
```

We can see that the revert operation has been written in the history, just as a usual commit.

Let see how it looks like inside our amazing file (it may be needed to close/reopen the file).

The monkey actually disappeared! :O



Work with branches

Within a given project, we can define several *branches*. Each branch will define different evolutions of the project. Git allows you to easily switch between those different branches, and to work in parallel on different *versions* of the same project. It is a central concept of a version control system.

Up to now, we worked on the default branch, which is by convention named *main*. This branch should be the most reliable, the most *stable*. A good practice is to **never work directly on the main branch**. We actually did not follow this rule up to now for simplicity. In order to keep the main branch stable, **each time we want to modify our project, we should create a new branch** to isolate our future changes, that may lead to break the consistency of the code.

Here is a representation of what is the current status of our project.

Fig. 7.103: We are on the *main* branch and we did 3 commits. The most recent commit of the branch is also called *HEAD*.

We will create a new branch, that we will call *develop*, with the following command

```
$ git branch develop
```

Then, we will *switch* to this branch, which means that from now on we will work on the *develop* branch.

```
$ git switch develop
```

It should look like this:

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git branch develop

dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git switch develop
Switched to branch 'develop'

dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (develop)
$
```

Notice that the name of the branch we are working on is displayed by Git Bash under brackets in light blue.

Within this branch, we will be very safe to try any modification of the code we like, because it will be completely isolated from the *main* one.

Let say that we now modify our file by adding some new animals (a bird and a mosquito), and committing at each time. Here is a representation of the new status of our project.

If we are happy with those two last commits, and we want to include them in the main branch, we will *merge* the *develop* branch into the *main* one, using the following procedure.

We first have to go back to the *main* branch. For that, we use

```
$ git switch main
```

Then, we tell Git to *merge* the *develop* branch into the current one, which is *main*

```
$ git merge develop
```

And we can now delete (with the *-d* option) the *develop* branch which is now useless.

```
$ git branch -d develop
```

We end up with a *main* branch that inherited from the last commits of the former *develop* one (RIP).

This procedure looks overkill at first sight on such a simple example, but we strongly recommend that you try to stick with it at the very beginning of your practice with Git. It will make you more familiar with the concept of branch and force you to code with a precise purpose in mind before doing any modification. Finally, the concept of branch will become much more powerful when dealing with the remote use of Git.

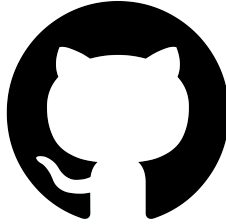
Local development workflow

To conclude, the local development workflow is as follow:

- Start from a clean repository.
- Create a new branch *develop* to isolate the development of my new feature from the stable version of the code in *main*. **Never work directly on the main branch!**
- Do modifications in the files.
- Test that the result is as expected.
- Do a commit.
- Repeat the 3 previous steps as much as necessary. **Try to decompose as much as possible any modification into very small ones.**
- Once the new feature is fully operational and tested, merge the *develop* branch into the *main* one.

Doing a commit is like saving your progression in a video game. It is a checkpoint where you will always be able to come back to, whatever you do after.

Once you will be more familiar with Git, you will feel very safe to test any crazy modification of your code!



Remote use of Git: GitHub

GitHub is a free cloud service that **allows anyone to have Git repositories on distant servers**. Such services allow their users to easily share their source code. They are an essential actors for the open-source development. You can find on GitHub such projects as the [Linux kernel](#), the software that runs [Wikipedia](#)... and last but not least: [PyMoDAQ](#)!

Other solutions exist such as [GitLab](#).

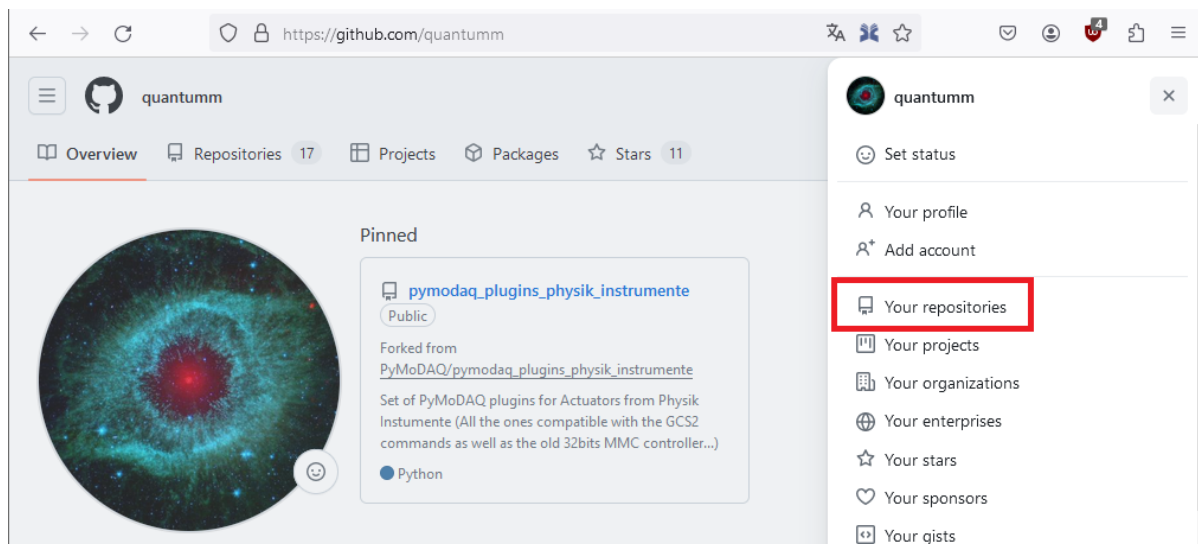
Create an account

First, we will need to create a personal account on GitHub. Please refer to the following tutorial to do so:

Create an account & raise an issue on GitHub

Create a remote repository

Once our profile is created, we go to the top right of the screen and click on the icon representing our profile.

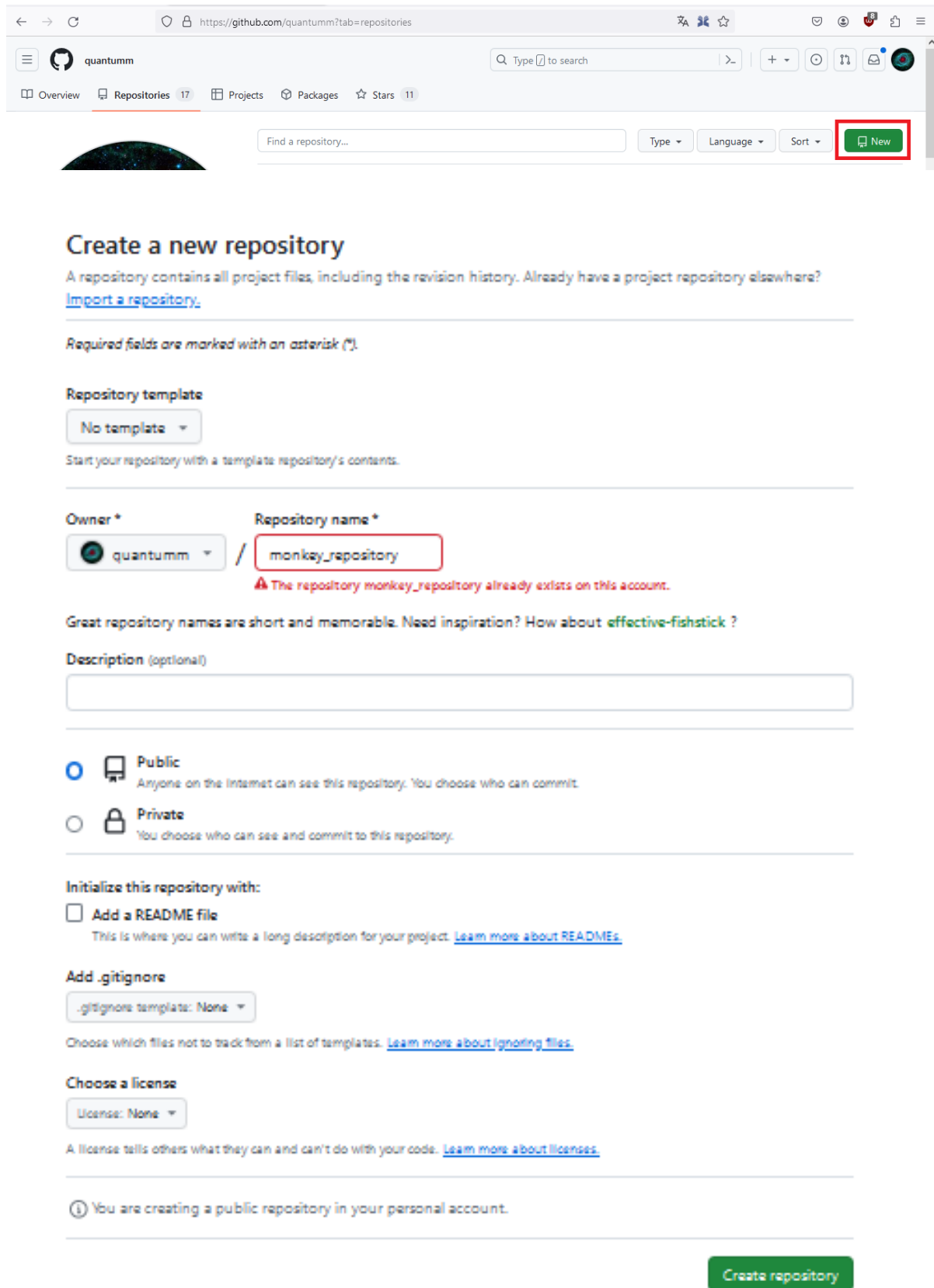


Let's create a remote repository.

Let's call it *monkey_repository* and click on *Create repository*.

Note: Note that we can create a *public* or a *private* repository. If we want the other users of GitHub to have access to the code that we will put in this repository, we will make it public. Otherwise we will make it private.

Let's stop here for a bit of vocabulary:



← → ↻ https://github.com/quantumm?tab=repositories

quantumm

Overview Repositories 17 Projects Packages Stars 11

Find a repository... Type Language Sort **New**

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner * quantumm / **Repository name *** monkey_repository

⚠ The repository monkey_repository already exists on this account.

Great repository names are short and memorable. Need inspiration? How about [effective-fishstick](#) ?

Description (optional)

Public ☒ Anyone on the Internet can see this repository. You choose who can commit.

Private ☐ You choose who can see and commit to this repository.

Initialize this repository with:

☐ Add a README file
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None ▾

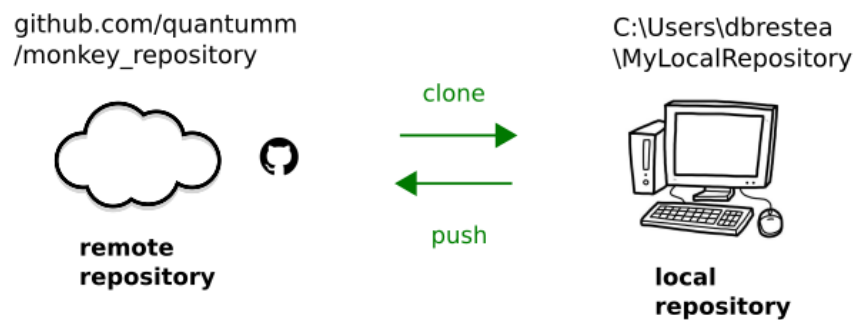
A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

① You are creating a public repository in your personal account.

Create repository

- Our **local repository** is the local folder that we created and configured to be followed by Git. Here it is our *MyLocalRepository* folder, that is stored on our local machine.
- We call **remote repository** the one that we just created. Its name is *monkey_repository* and its Git address is *https://github.com/Fakegithubaccountt/monkey_repository.git*.
- When we will talk about **pushing**, we will mean that we upload the state of our local repository to the remote repository.
- When we will talk about **cloning**, we will mean that we downloaded the state of the remote repository to a local repository.

All this is summed up in the following schematic.



Authentication to GitHub with an SSH key

To get authorized by GitHub to interact with our remote repository, we will need to authenticate to it. Hopefully, it will not let anyone push what he wants on this repository! We have to prove him that we own the repository. The authentication is a bit more complicated than using a password, we will use the *SSH protocol*. No worries, everything is explained step by step in the following tutorial:

Authenticate to GitHub with an SSH key

Push our local repository to GitHub

We started this tutorial from a local folder, and then created a remote repository on our GitHub account. For now the latter is empty. What we will do now is to push the content of our local repository to our remote repository.

Note: Note that it is not obvious that we will always work this way. Most of the time, we will start by cloning a remote repository to our local machine.

With the following command, we tell Git that our local repository (the folder where we are executing the command) from now on will be connected to the remote repository that we just created on GitHub. The latter is called *origin* by default. Be careful to be at the root of our local repository to execute the following command:

```
$ git remote add origin <the Git address of our remote repository>
```

Note: The Git address of a repository follows the naming convention `https://github.com/GitHub_username/repository_name.git`

With the next command, we will check that everything is as expected. We call for information about the remote repository.

```
$ git remote -v
```

It should look like this:

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git remote add origin https://github.com/quantumm/monkey_repository.git

dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git remote -v
origin https://github.com/quantumm/monkey_repository.git (fetch)
origin https://github.com/quantumm/monkey_repository.git (push)
```

This is all good. The first line, ending with *fetch*, means that when we will ask to update our local repository (with a *pull* command, we will see that latter), it will call the *origin* repository. The second line, ending with *push*, means that when we will ask to update the remote repository with the work we have done locally, it will go to *origin*.

Let us try to push our repository!

```
$ git push -u origin main
```

Note: Notice that when we push, we push a specific branch, which is *main* here.

It should look like this:

```
dbrestea@iram-fb-004873 MINGW64 ~/MyLocalRepository (main)
$ git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 938 bytes | 312.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/quantumm/monkey_repository.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

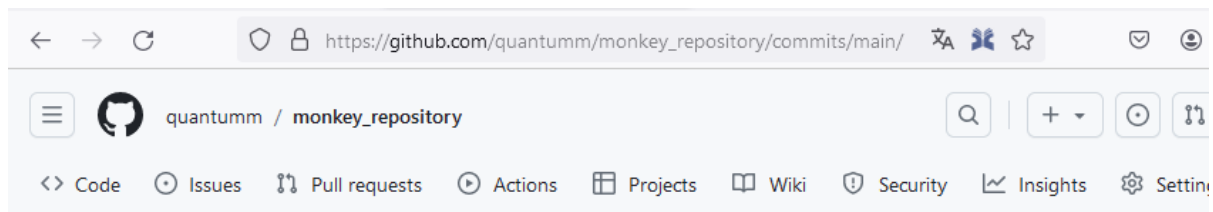
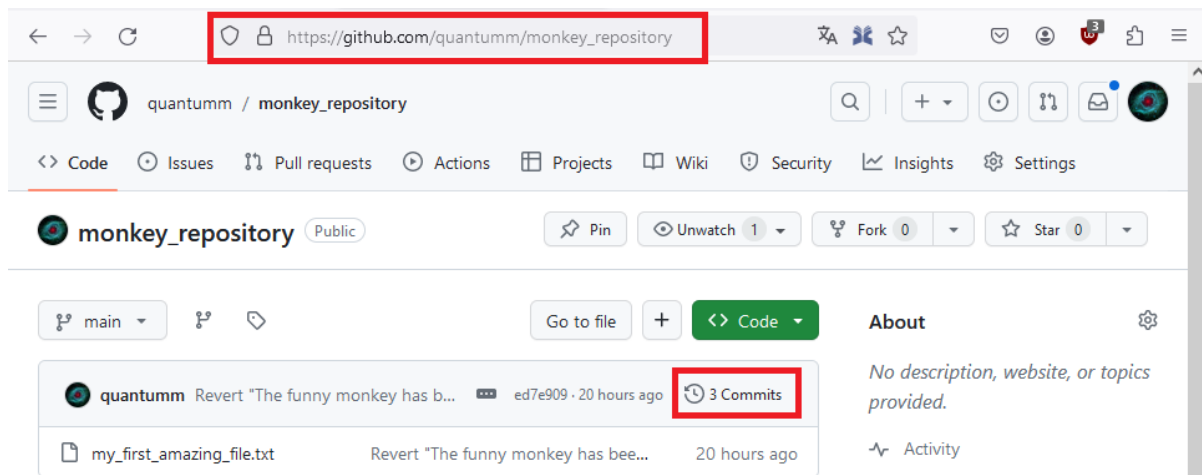
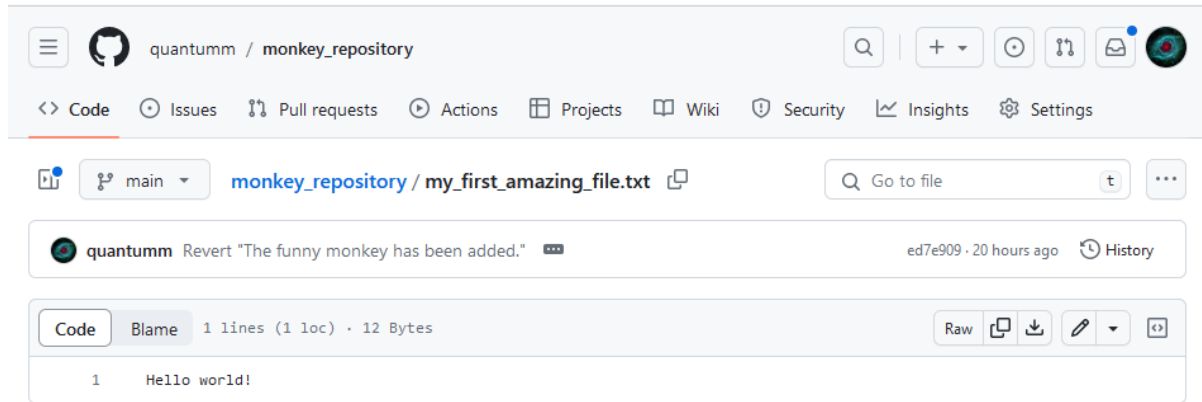
Our file is online!

But it is not like we just store a file on a server, we also have access to all the history of the commits.

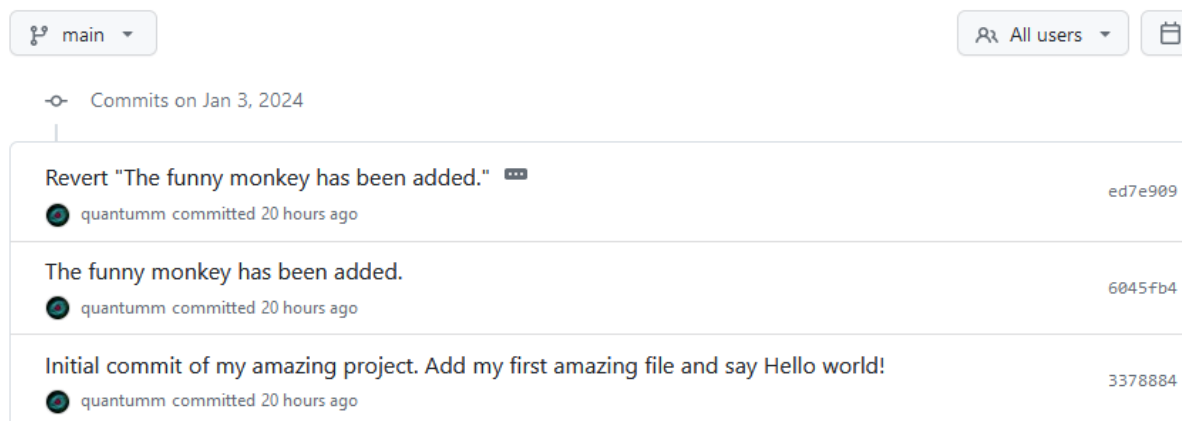
Here they are.

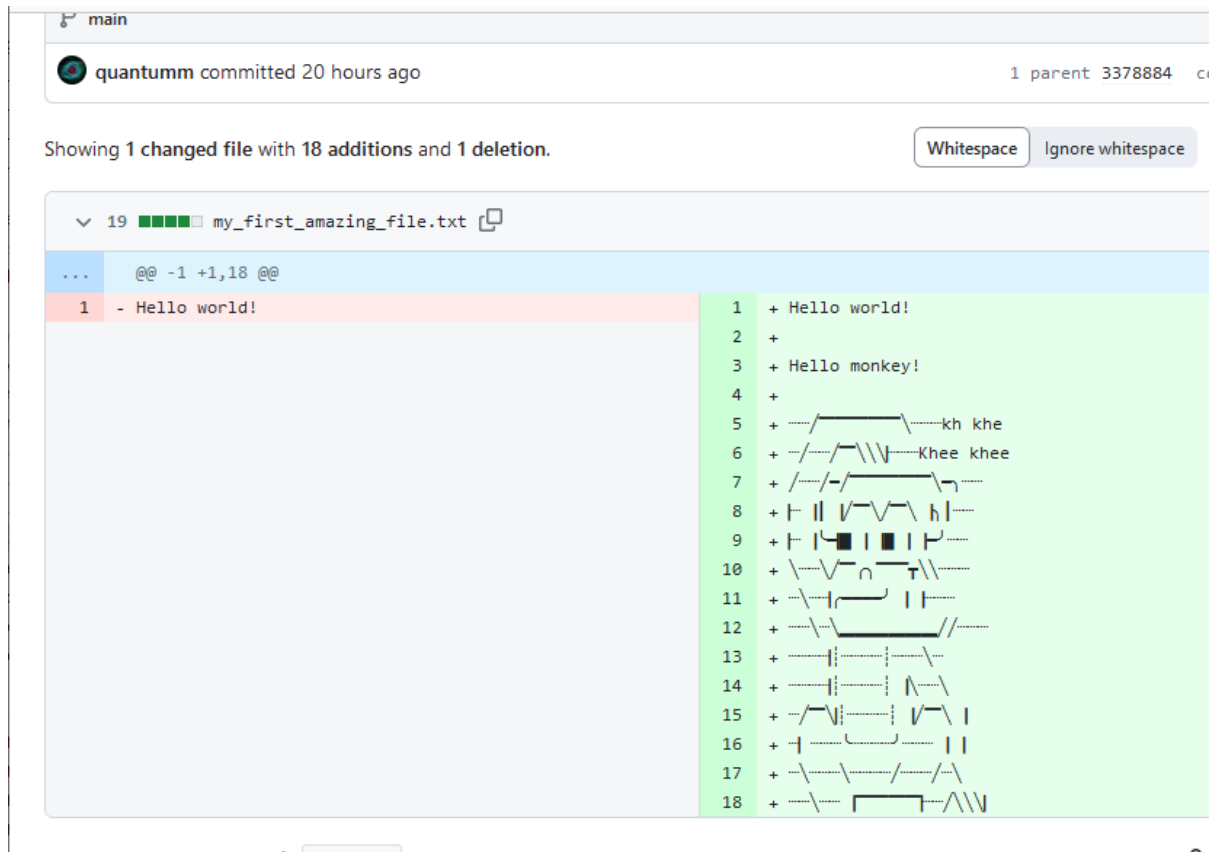
Let's click on the second commit *The funny monkey has been added*.

Here he is!



Commits





We see that the GitHub website provides an equivalent to what we see in the terminal. The advantage is that now we can access it from any computer connected to internet!

Finally, the development workflow is as follow:

- Do modifications in the file on our local repository.
- Test that the result is as expected.
- Do a commit.
- We can repeat the previous steps several times.
- At the end of the day, we push all our commits to our remote repository.

Now, our remote repository should always be our reference, and not our local version anymore!

The latest version of our code must be stored on the server, not locally. Once our push is done, we can safely delete our local folder. We will be able to get our code back at the latest version at any time from any computer, thanks to the *clone* command.

If you have further questions about the management of remote repositories, you can refer to this documentation:

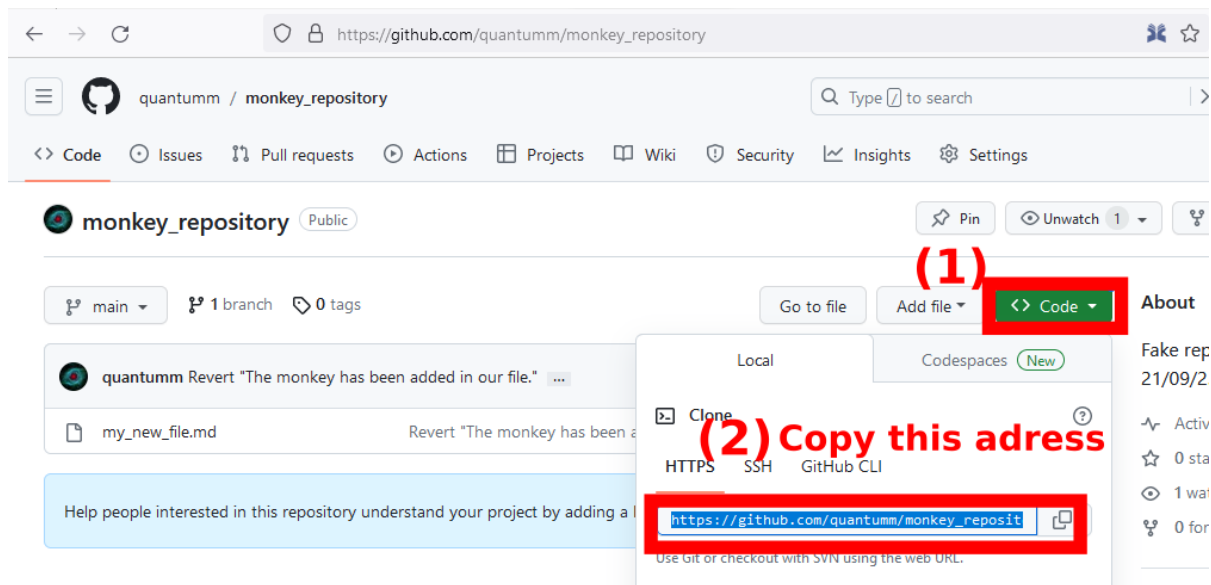
[Managing remote repositories \(github.com\)](https://github.com)

The `clone` command

Ok so let's do it, let's delete our local folder *MyLocalRepository*. We will convince ourself that we can easily find it back.

Since our work is now stored on a GitHub server, it is not a problem even if our computer goes up in smoke. We can get it back with the `clone` command.

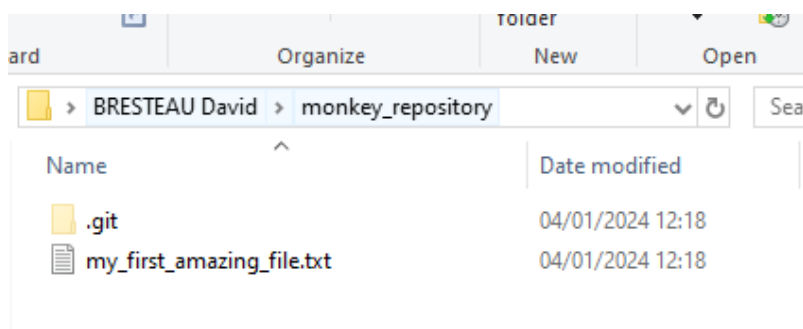
First, copy the Git adress of the repository



Then, at our home location, we execute the command

```
$ git clone <the Git address of our remote repository>
```

```
dbrestea@iram-fb-004873 MINGW64 ~
$ git clone https://github.com/quantumm/monkey_repository.git
Cloning into 'monkey_repository'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
```



We found our work back!

Note: Notice that by default, the *clone* command will create a folder with the same name as the remote repository, but this is not mandatory. If you want another name for your local repository you can use `$ git clone <repository url> <your folder name>`.

Notice that when we clone a repository, we do not need anymore the *init* command. We do not need either to configure the address of the remote repository, Git already knows where to took it from.

We can follow this procedure for any public repository on GitHub, which allows us to download basically all the open-source codes in the world!

Git in practice: integration within PyCharm

We now master the basics of using Git with the command line (CLI), and it is like this that we get the best control of Git. But we should know that there are several graphical user interfaces (GUI) that can ease the use of Git in the daily life, such as [GitHub Desktop](#) if we are working with Windows.

However, we will rather recommend to use the direct integration within your favorite Python IDE, because it does not require to download another software, and because it is cross platform. We will present the practical use of Git with [PyCharm](#). The *Community Edition* is free of charge and has all the functionalities that we need.

Link our GitHub account to PyCharm

As a first step, we should authorize PyCharm to connect to our GitHub account. We recommend to use a token. This way we will not have to enter a password each time PyCharm needs to connect to GitHub. The procedure is described in the following documentations:

[PyCharm & GitHub \(jetbrains.com\)](#)

[PyCharm Integration with GitHub \(medium.com\)](#)

Note: It seems like SSH connection is only for the *Professional* version of PyCharm, which is charged.

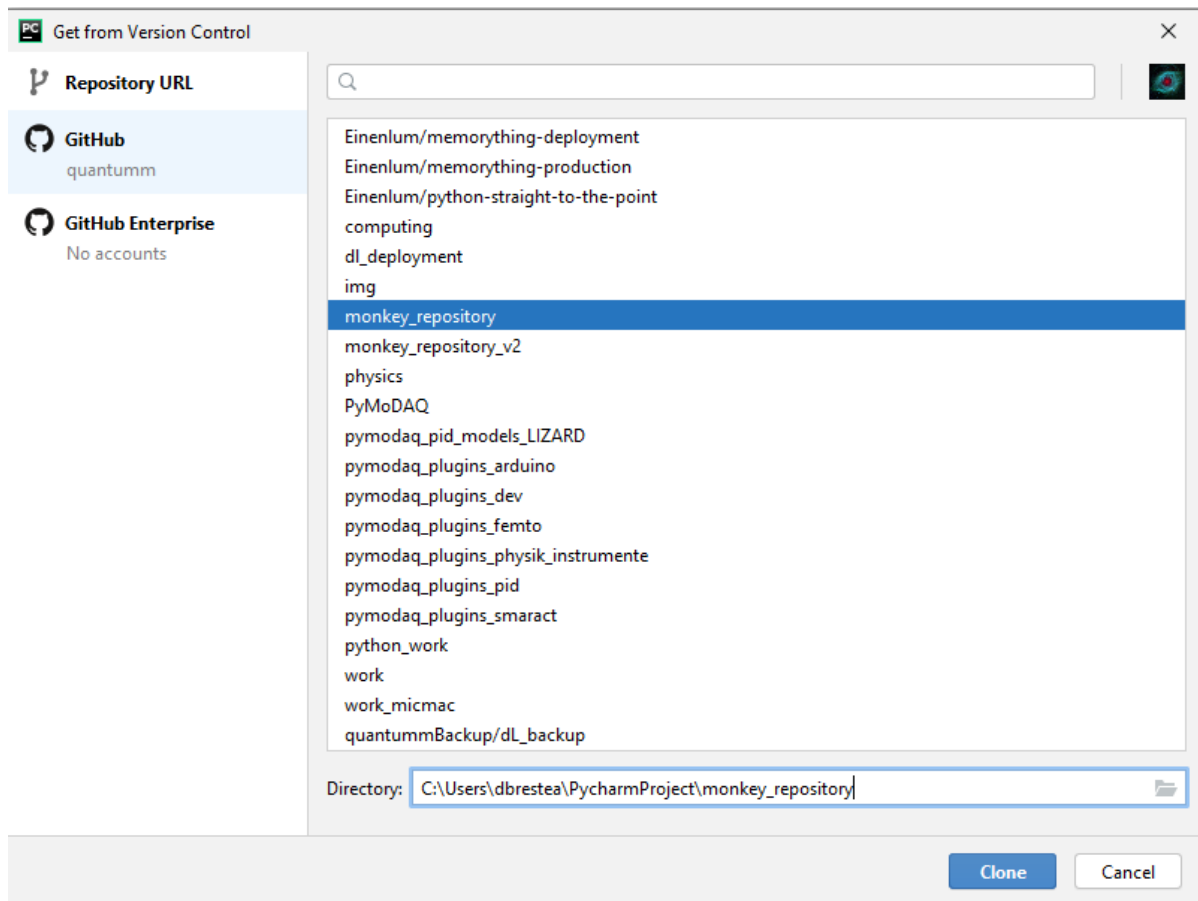
Clone a project

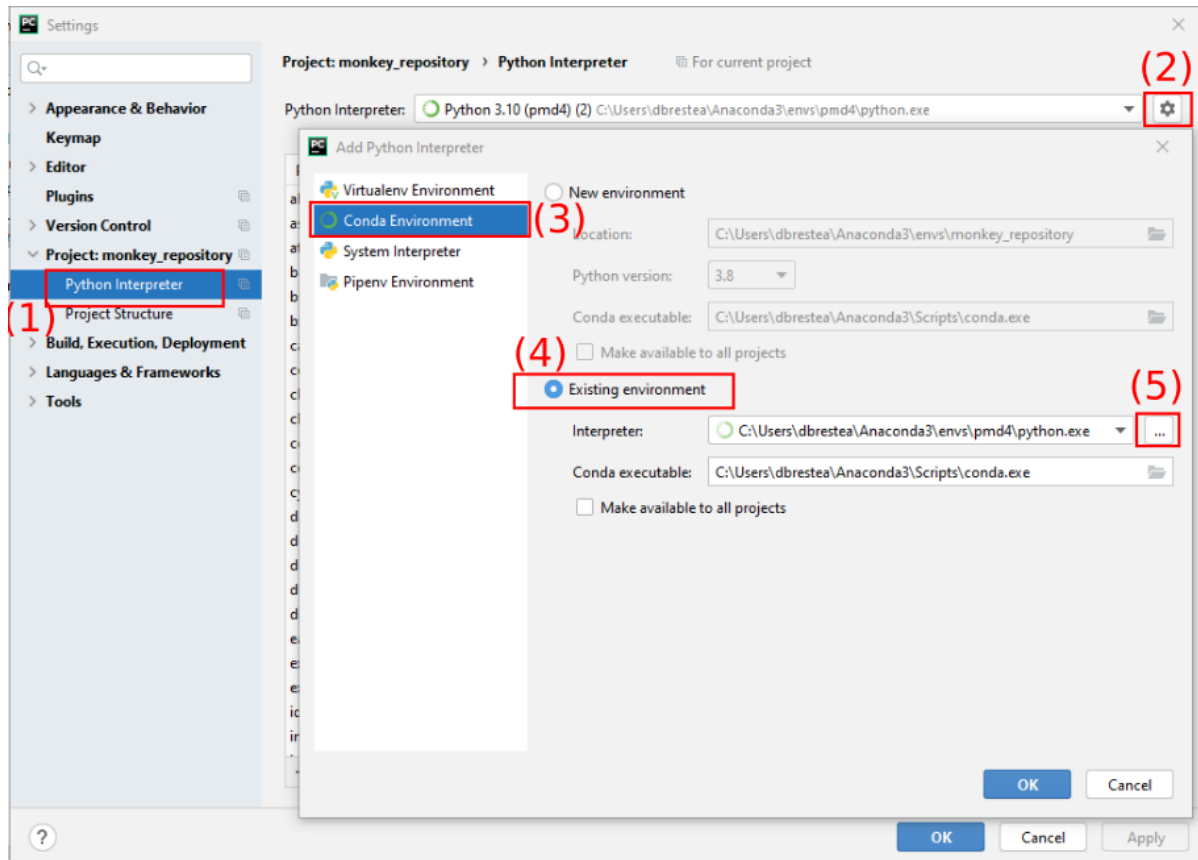
We first clone the *monkey_repository* from our GitHub account. Go to Git > Clone..., select the remote repository and a local folder where the files will be saved (it does not matter where we decide to save locally the repository).

Configure our Python environment

Once the remote repository has been cloned, we have to configure our environment. Go to File > Settings... and select an existing Conda environment (here it is called *pmd4*).

Note: Documentation about setting up a new Python environment can be found here: [PyMoDAQ installation](#).





Create a new branch

Here are the main important places on the PyCharm interface to manage Git.

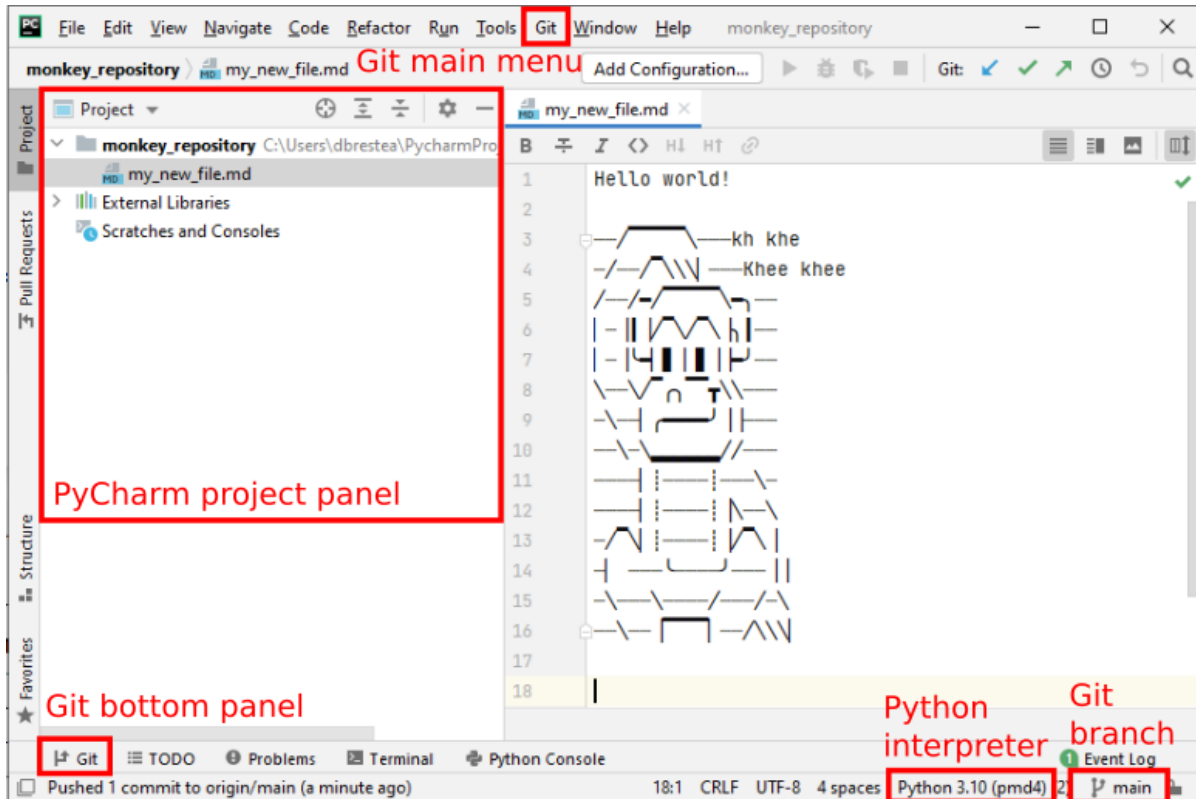
We will follow our best practices and create a new local branch before modifying the files in the repository. To do so we click on the *Git branch button* (see screenshot above) and create a new branch that we call *develop*.

Diff, commit and push

Let's now add a bird in the file.

Then go to `Git > Commit...` It will open a window that allows us to easily see the files that have been modified. If we right click on *my_new_file.md* and select *Show diff*, we will see the difference between the two versions of the file, just as with the command line, but with a more evolved interface.

If we are happy with that, we can close this window and *Commit & Push* our changes with the corresponding button.



Add a file

Adding a file is also very easy since you just have to *Paste* it in the right folder within the *Project* panel of PyCharm: right click on the corresponding folder and select *Paste* or *New file* if you start from an empty one.

It will automatically ask us if we want Git to track the new file.

Log

If we open the *Git bottom panel* we can have information about the local and remote branches, and the history of the commits.

Conclusion

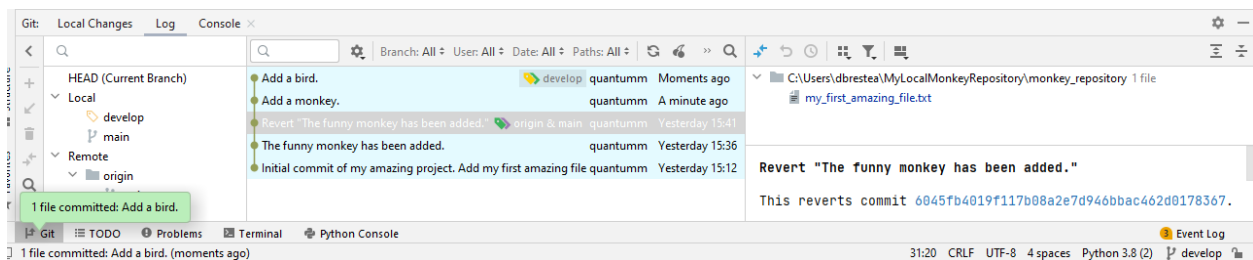
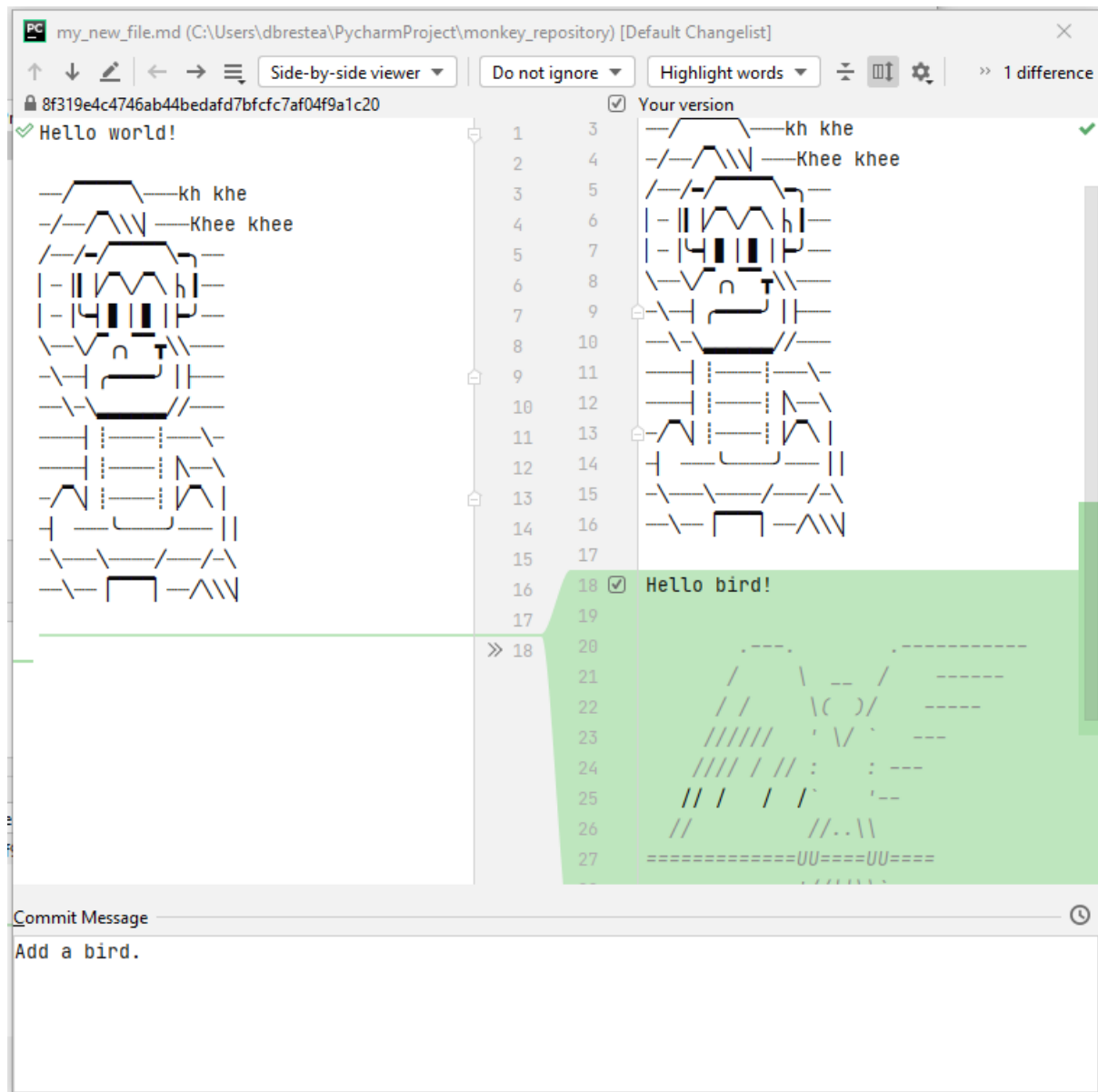
We now master the basics of the worldwide standard for code development! Following those guidelines, we will code more efficiently. Git is appropriate for any (descent) language (not Word or Labview!). It is an indispensable tool if we want to share our code with colleagues and not reinvent the wheel. Git is one of the reasons why we will make better acquisition programs with PyMoDAQ than with Labview ;)

If you want to go further and learn how to contribute to any external open-source code, we invite you to pursue with the tutorial

How to contribute to PyMoDAQ's code?

Finally, here are a few external ressources:

The YouTube channel of Grafikart (in French)

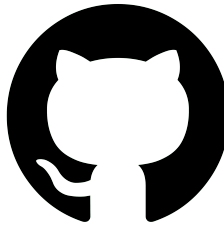


The course of OpenClassroom (in English)

The Pro Git book (in English). Exhaustive and painful. You will probably not need it!

If you have any remarks regarding this tutorial, please do not hesitate to *raise an issue* or write an email to the author.

Author email	david.breseau@cea.fr
Last update	december 2023
Difficulty	Intermediate



Authenticate to GitHub with an SSH key

In general, when we need to authenticate to a website, we will provide a password. Since quite recently, it is not possible to make our local Git connect to GitHub with a password. It is now mandatory to connect with the SSH protocol for security reasons. We thus have to follow this quite obscure procedure (it is not so bad!). After overcoming this little difficulty, the reward will be that we will not have to enter any password anymore to interact with GitHub!

Prerequisite

To follow this tutorial, you should already have a GitHub account and Git installed on your local machine. If it is not the case, please start with the following tutorials:

Create an account & raise an issue on GitHub

Basics of Git and GitHub

What is SSH?

SSH, for Secure SHell, is a protocol that permits to connect to distant servers safely. Underlying it uses *public-key cryptography* to implement a secure connection between our local machine (the client) and GitHub (the server). Each of the two parts will have a *public* and a *private key*. Those *keys* are basically big numbers stored in files.

If you want to know more about SSH, you can read this documentation: [About SSH \(GitHub\)](#)

How to make a secure connection with SSH?

Let's take a big breath, we do not need to know what is happening in details! We will just follow blindly the procedure that is proposed by GitHub. Basically there are 3 steps:

- We have to generate our private and public SSH keys (our *SSH key pair*). Our *private key* will be kept on our local machine.
- We then have to add our private key to the *ssh-agent*. Whatever the *ssh-agent* is... let say it means that we tell SSH to take this new private key into account and manage it.

- Finally, we will have to add our *public key* to our GitHub account.

Let's go!

Generate our SSH key pair

Let's open a *Git Bash* terminal.

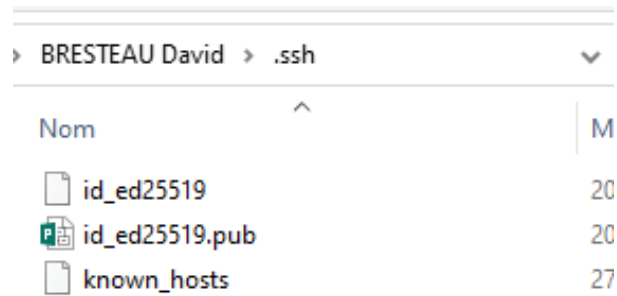
Note: If you are working with Windows, *Git Bash* should be installed on your machine. If it is not the case, follow the procedure that is described in the tutorial [Basics of Git and GitHub](#). If you are working with Ubuntu, just use a standard terminal.

Copy-paste the following command that will generate our key pair. We should replace the email address by the one that is linked to our GitHub account.

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```

Press *Enter* to every question that is prompted.

We now have several files that are stored in a *.ssh* folder that have been created at our home (C:\Users\dbretea). If you do not see the *.ssh* directory maybe you need a Ctrl + H to show the hidden folders.



The *id_ed25519.pub* file contains our public key. The *id_ed25519* file contains our private key. We should never reveal the content of the latter, it must stay only on our local machine.

Add our private key to the ssh-agent

Now that we have our key pair, we must tell SSH to manage this key, using the following command

```
$ ssh-add ~/.ssh/id_ed25519
```

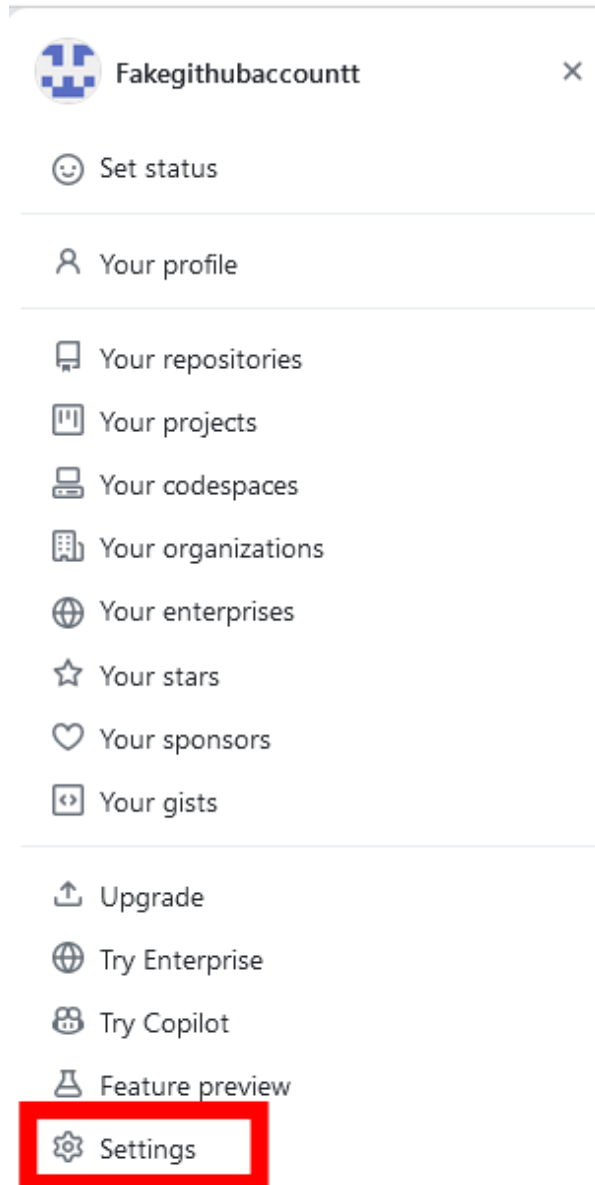

Add our public key to our GitHub account

We will now copy the content of our public key with the following command, which is equivalent to opening the file and copying its content to the clipboard

```
$ clip < ~/.ssh/id_ed25519.pub
```

Note: Notice that we use the public key here by taking the file with the *.pub* extension.

We now have to paste it in our GitHub settings.



And paste the key in the form

Finally, press the *Add SSH key* button. We are done ;)

This section has been inspired by those documentations:

SSH keys (2)

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Authentication Keys

Key name	Key type	Added on	Used	Permissions	Actions
ssh_key_github	SSH	Sep 20, 2023	Never used	Read/write	Delete

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH problems](#).

GPG keys

New GPG key

(1) SSH and GPG keys

Add new SSH Key

Give a title like "Work computer"

Key type: Authentication Key

Paste the key here

Key: Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'

Add SSH key

Generating a new SSH key and adding it to the ssh-agent (GitHub)

Adding a new SSH key to your GitHub account (GitHub)

Concluding remarks

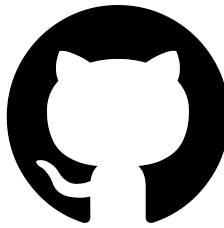
We are now ready to easily and safely interact with our remote repositories on GitHub!

Note that this procedure must be done again each time you want to interact with your GitHub repositories with a different machine.

If you have any remarks regarding this tutorial please do not hesitate to *raise an issue* or write an email to the author.

7.5.2 How to modify existing PyMoDAQ's code?

Author email	david.bresteau@cea.fr romain.geneaux@cea.fr
Last update	january 2024
Difficulty	Intermediate



In this tutorial, we will learn how to propose a modification of the code of PyMoDAQ. By doing so, you will learn how to contribute to any open-source project!

Prerequisite

We will suppose that you have followed the tutorial

Basics of Git and GitHub

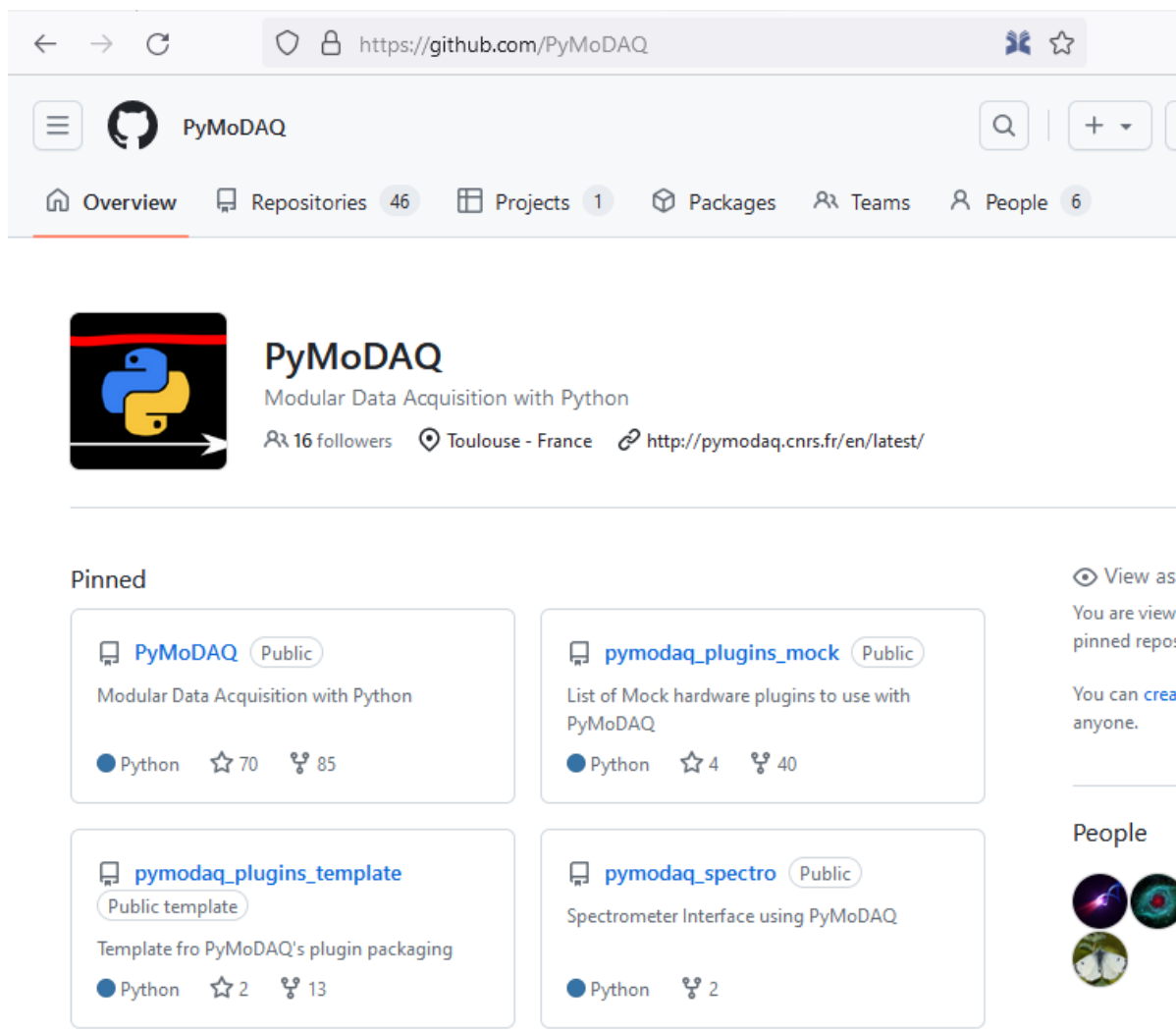
In the latter, we presented how to deal with the interaction of a *local repository* with a *remote repository*. Up to now we just worked on our own. In the following we will learn how to contribute to an external project like PyMoDAQ!

The PyMoDAQ repositories

Let's now go to the [PyMoDAQ GitHub account](#).

There are a lot of repositories, most of them correspond to *Python packages*. Briefly, there is:

- The **PyMoDAQ repository**: this is the core of the code, you cannot run PyMoDAQ without it.
- The **plugins' repositories**: those repositories follow the naming convention `pymodaq_plugins_<name>`. Most of the time, `<name>` corresponds to the name of an instrument supplier, like *Thorlabs*. Those are optional pieces of code. They will be useful depending on the instruments the final user wants to control.



The screenshot shows the GitHub repository page for PyMoDAQ. The browser address bar displays `https://github.com/PyMoDAQ`. The repository name **PyMoDAQ** is shown with its logo, a Python logo with a red arrow. The description is "Modular Data Acquisition with Python". It has 16 followers, is located in Toulouse - France, and has a website link `http://pymodaq.cnrs.fr/en/latest/`.

The **Pinned** section displays four repositories:

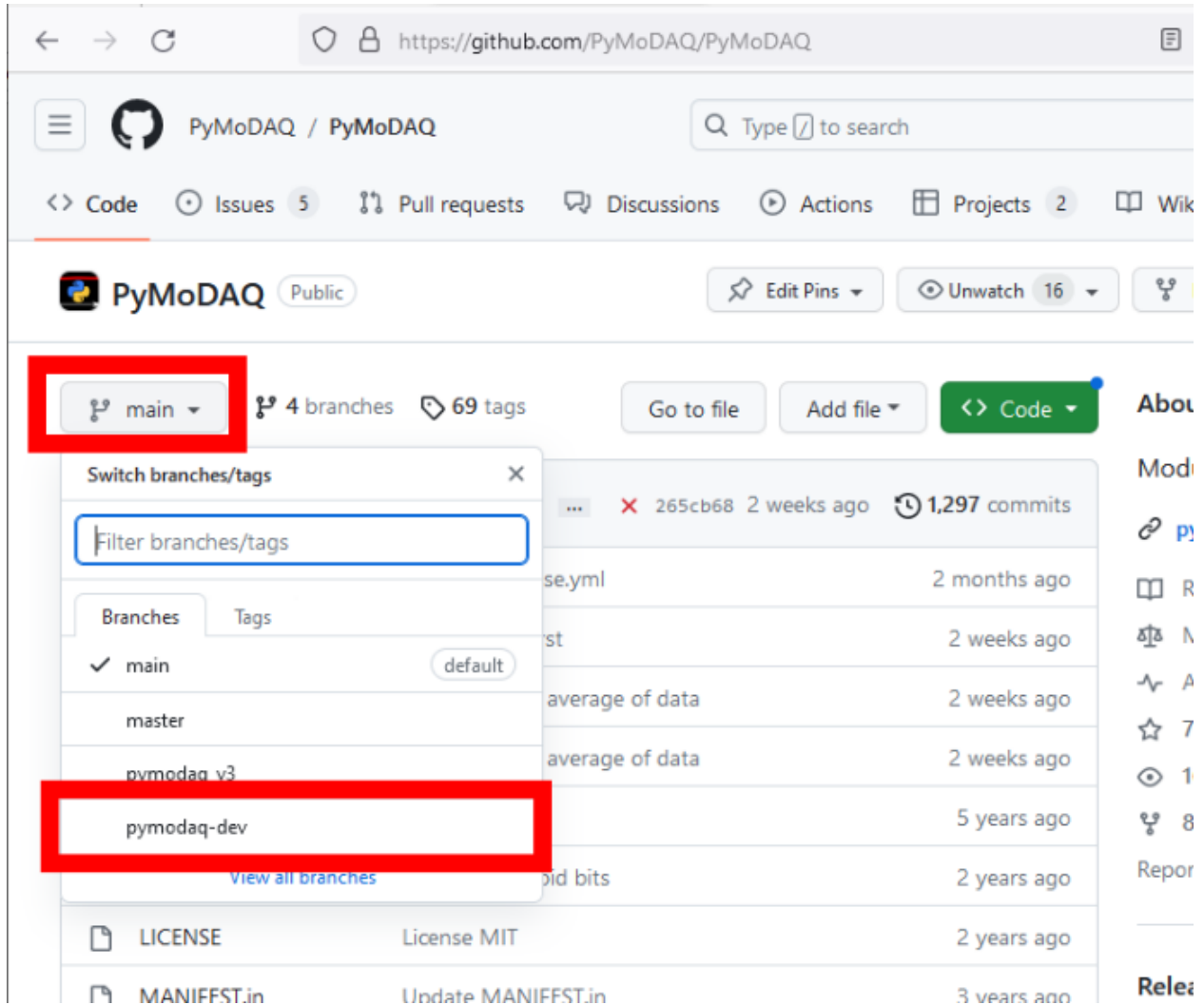
- PyMoDAQ** (Public): Modular Data Acquisition with Python. 70 stars, 85 forks.
- pymodaq_plugins_mock** (Public): List of Mock hardware plugins to use with PyMoDAQ. 4 stars, 40 forks.
- pymodaq_plugins_template** (Public template): Template for PyMoDAQ's plugin packaging. 2 stars, 13 forks.
- pymodaq_spectro** (Public): Spectrometer Interface using PyMoDAQ. 2 forks.

On the right, the **View as:** section indicates the user is viewing pinned repositories. The **People** section shows three profile pictures.

PyMoDAQ branches

Let's go to the PyMoDAQ repository.

Note: Be careful not to confuse the PyMoDAQ *GitHub account* and the *repository*.

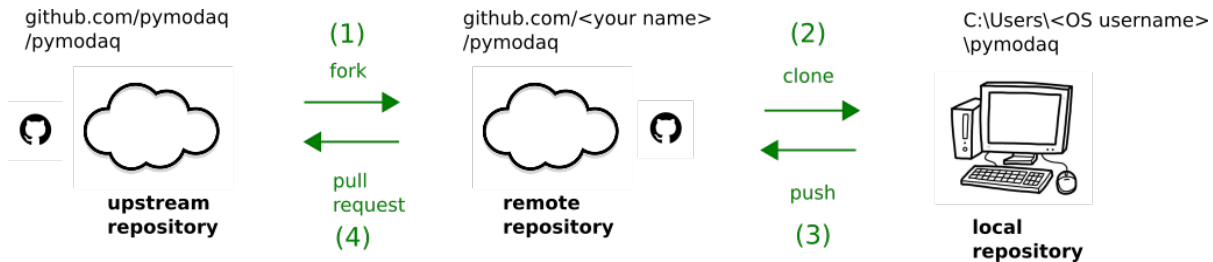


There are several branches of the PyMoDAQ repository. Branches are used to prepare future releases, to develop new features or to patch bugs, without risking modifying the stable version of the code. The full branch structure is described at length *in the Developer's guide*. For our purposes here, let us just mention the two most important branches:

- **the stable branch.** It is the present state of the code. When you install PyMoDAQ with pip, it is this version of the code that is downloaded.
- **The development branch.** It is *ahead* of the main branch, in the sense that it contains more recent commits than the main branch. It is thus the future state of the code. This is where the last developments of the code of PyMoDAQ are pushed. When the developers are happy with the state of this branch, typically when they finished to develop a new functionality and they tested it, they will merge the develop branch into the main branch, which will lead to a new *release* of PyMoDAQ.

How to propose a modification of the code of PyMoDAQ?

Compared to the situation in the *Basics of Git and GitHub* tutorial, where we had to deal with our *local repository* and our *remote repository*, we now have to deal with an external repository on which we have no right. This external repository, which in our example is the PyMoDAQ one, is called the **upstream repository**. The workflow is represented the schematic below and we will detail each step in the following.



(1) Fork the upstream repository

Note: In the screenshots below, the stable and development branches are called *main* and *pymodaq-dev*. This naming scheme is now deprecated. Branch names now correspond to the current PyMoDAQ versions. For instance, if the current stable version is 5.6.2, the stable branch will be called *5.6.x* and the development branch will be called *5.7.x_dev*.

While we are connected to our GitHub account, let's go to the PyMoDAQ repository and select the *pymodaq-dev* branch. Then we click on the *Fork* button.

This will create a copy of the PyMoDAQ repository on our personal account, it then become our remote repository and **we have every right on it**.

Every modification of the code of PyMoDAQ should first go to the *pymodaq-dev* branch, and not on the *main* branch. The proper way to propose our contribution is that we create a branch from the *pymodaq-dev* branch, so that it will ease the integration of our commits and isolate our work from other contributions.

We create a branch *monkey-branch* from the *pymodaq-dev* branch.

(2) Clone our new remote repository locally

We will now clone our remote repository locally.

Open PyCharm. Go to *Git > Clone...* and select the *PyMoDAQ* repository, which correspond to our recent fork.

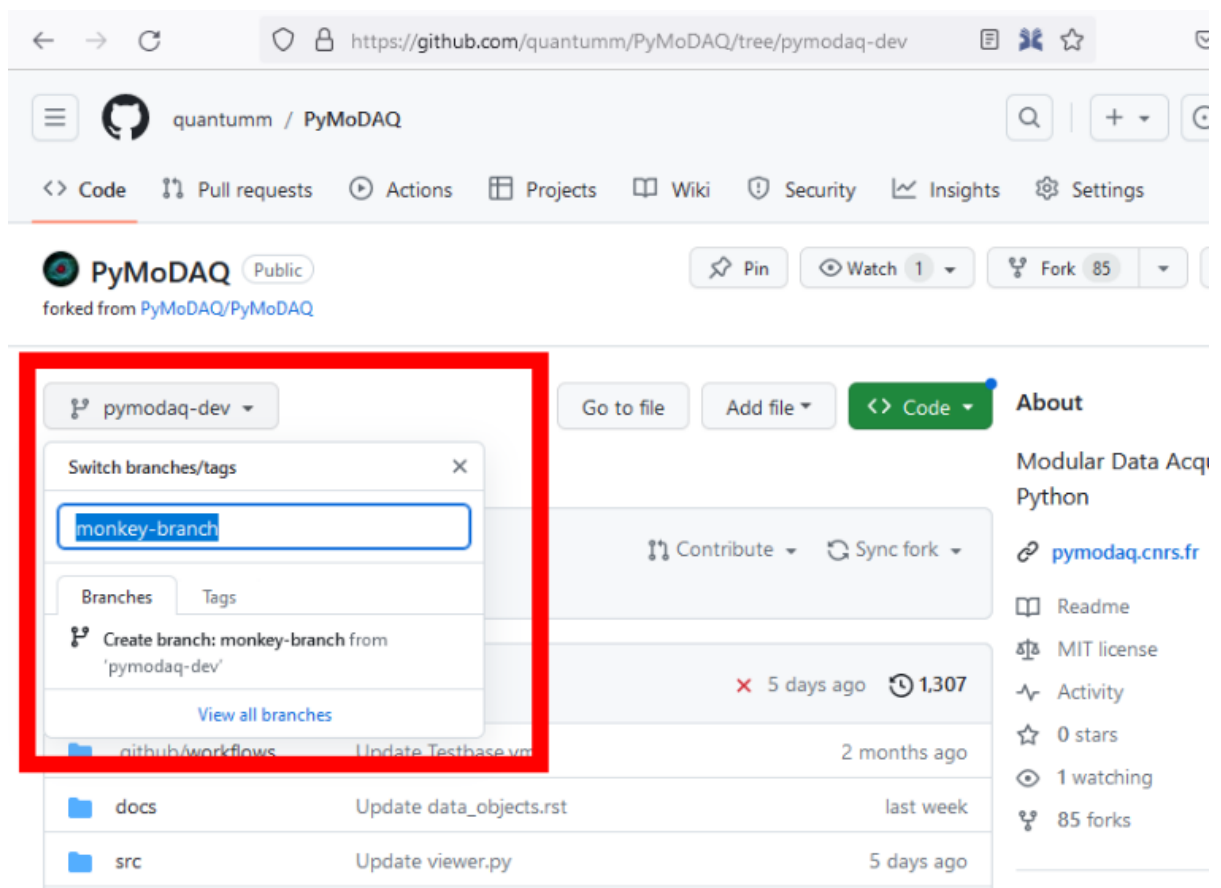
Note: Here we put the local repository inside a *PyCharmProject* folder and called it *PyMoDAQ*, but you can change those names if you wish.

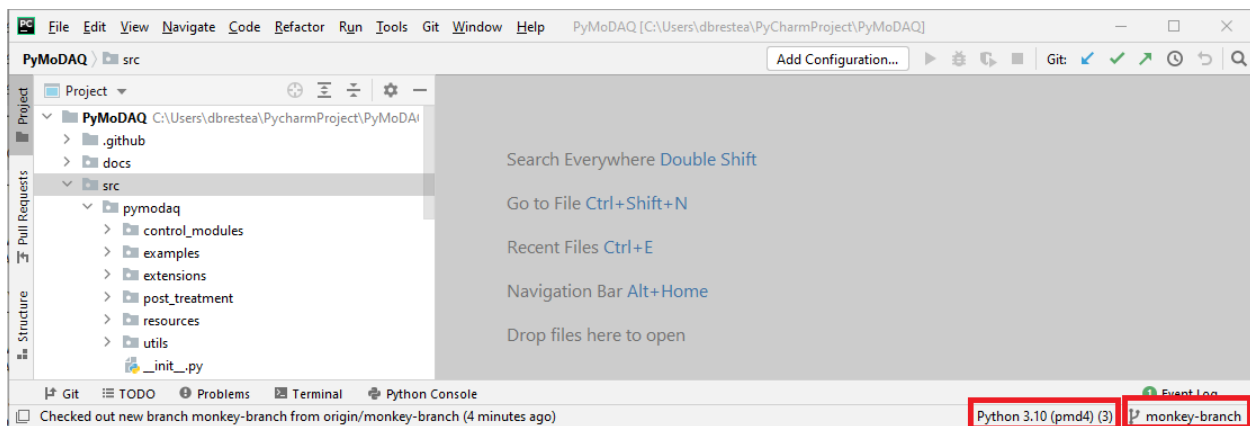
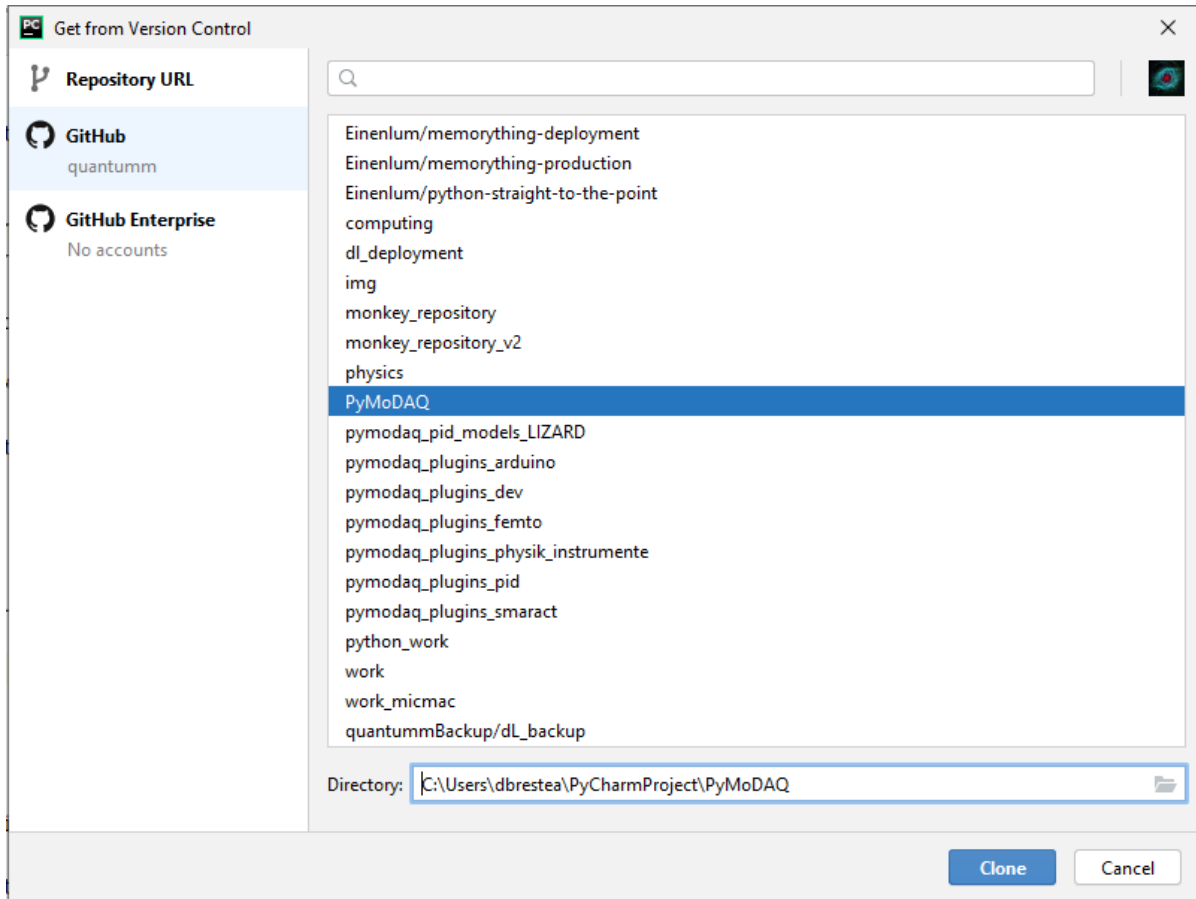
We configure PyCharm so that we have the good Python interpreter and we choose the *monkey_branch* of our remote repository.

The image consists of two screenshots of a GitHub repository page.

Top Screenshot: The repository is **PyMoDAQ / PyMoDAQ**. The **Code** tab is selected. The **pymodaq-dev** branch is selected (1). The **Fork** button is highlighted (2). The repository is public and has 85 forks and 16 watchers. The **About** section shows the repository is for **Modular Data Acquisition Python** and provides a link to pymodaq.cnr.fr. The **Files** section shows a commit by **seb5g** titled **Update viewer.py** 5 days ago with 1,307 changes. The commit details show updates to `.github/workflows`, `docs`, `src`, and `tests`.

Bottom Screenshot: The repository is **quantumm / PyMoDAQ**, which is a fork of **PyMoDAQ / PyMoDAQ**. The URL <https://github.com/quantumm/PyMoDAQ/tree/pymodaq-dev> is highlighted. The repository is public and has 85 forks and 1 watcher. The **About** section shows the repository is for **Modular Data Acquisition Python** and provides a link to pymodaq.cnr.fr. The **Files** section shows a commit by **seb5g** titled **Update viewer.py** 5 days ago with 1,307 changes. The commit details show updates to `.github/workflows` and `docs`.

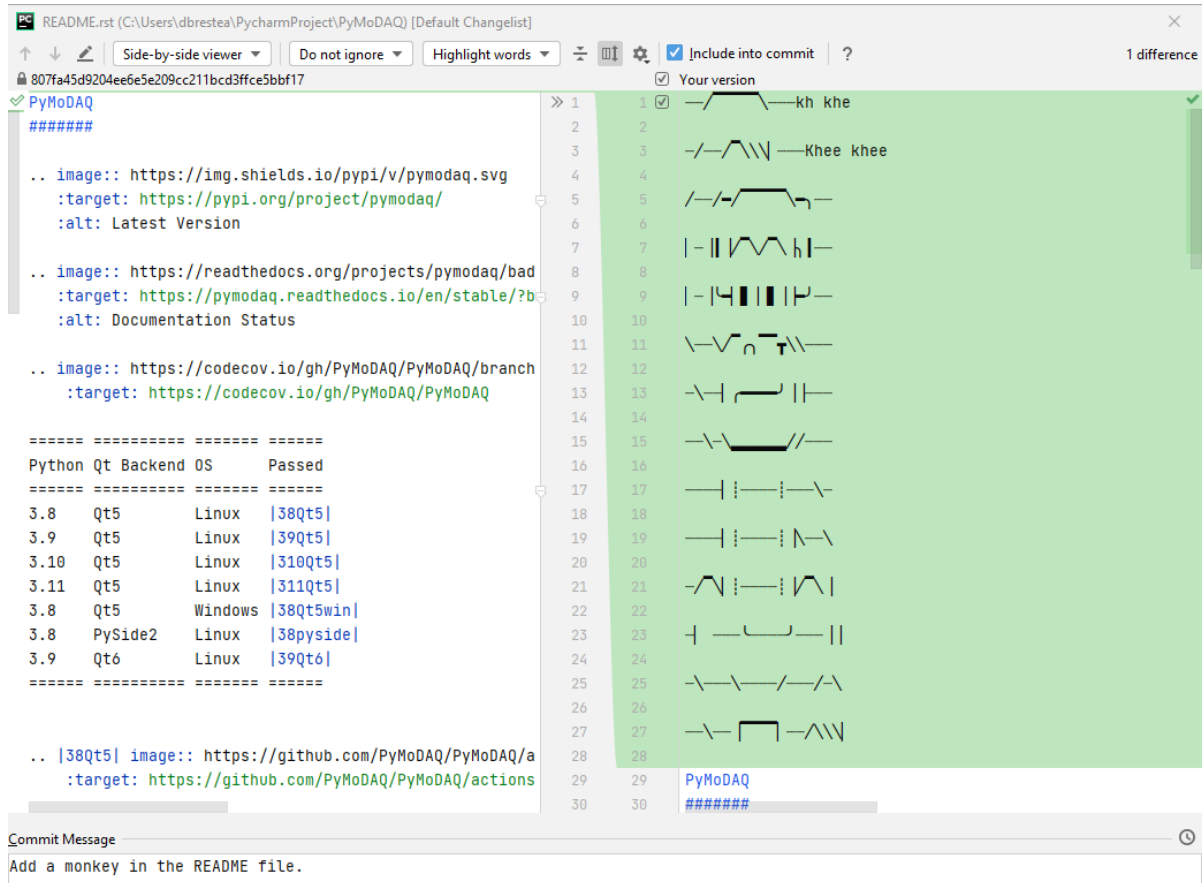




(3) Do modifications and push

We now have the PyMoDAQ code on our local machine. We will put the monkey into the README.rst file at the root of the PyMoDAQ package. This file is the one that is displayed at the home page of a GitHub repository.

We can now go to *Git > Commit...*, right click on the file and *Show Diff*.



If we are happy with our modifications, let's add a commit message and click *Commit and Push*.

This is the result on our remote repository.

We will now propose this modification, so that the monkey would appear at the front page of the PyMoDAQ repository!

(4) Pull request (PR) to the upstream repository

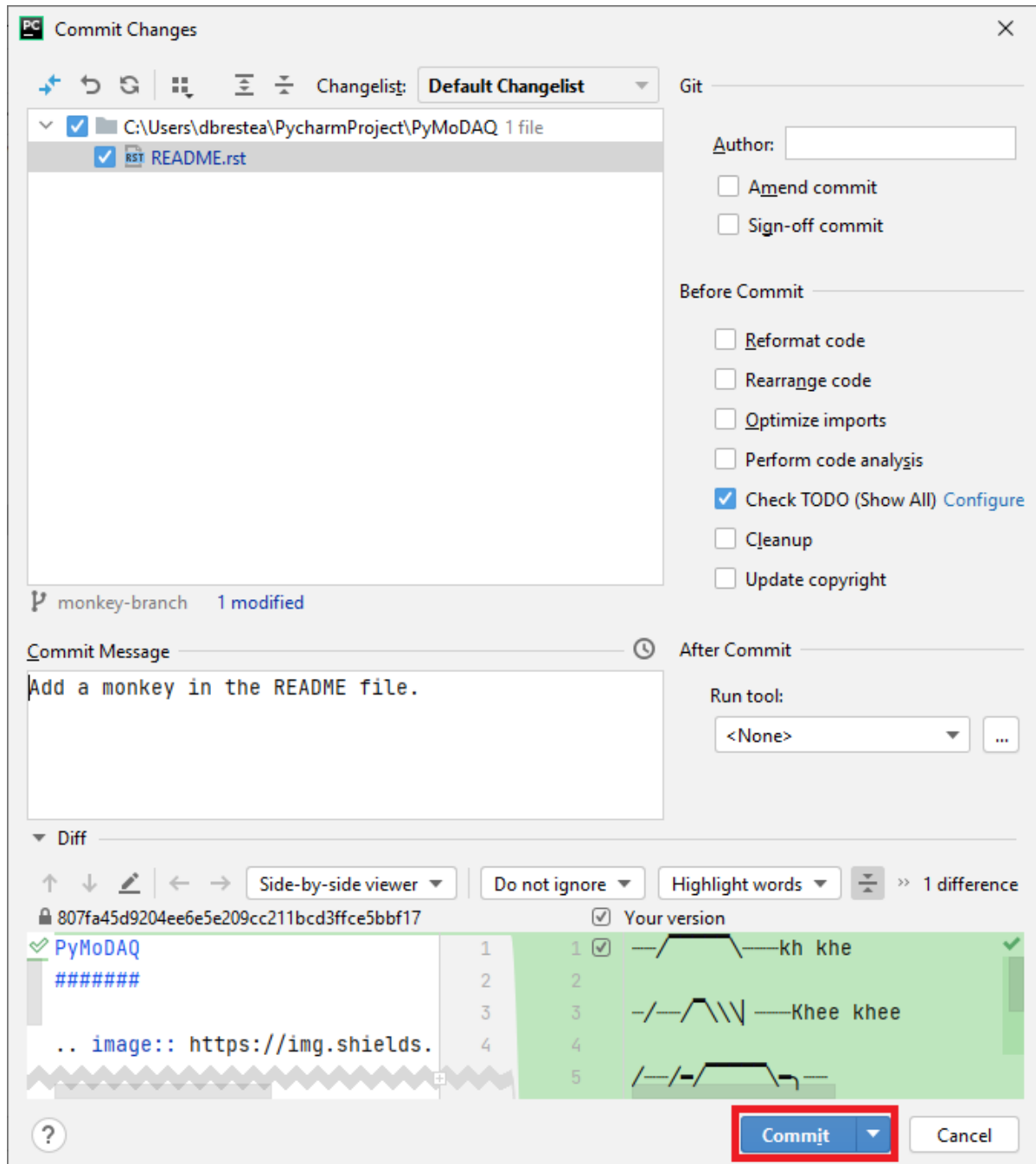
We can be very proud of our modification, but of course, this will not be implemented directly, we will need the agreement of the owner of the PyMoDAQ repository.

Opening a pull request is proposing a modification of the code to the owner of the upstream repository.

This can be done through the GitHub website, at the location of our repository. Either click to *Compare & pull request* or to the *Pull requests* tab and *New pull request*.

Be careful to properly select the branch of our repository and the branch of the upstream repository, and then *Send*.

That's it! We now have to wait for the answer of the owner of the upstream repository. Let's hope he will appreciate our work! We can see the status of our PR on the PyMoDAQ repository home page, by clicking on the *Pull requests* tab. There a discussion will be opened with the owner of the repository.



iconico

new release

5 years ago

pyproject.toml

Making sure nested dict in the config are properly updated

5 months ago

readthedocs.yml

remove issue with creating a folder using asmin rights

9 months ago

Language

Python

README.rst

kh khe

Khee khee

PyMoDAQ

pypi v4.0.11

docs passing

codecov 45%

Python	Qt Backend	OS	Passed
3.8	Qt5	Linux	3.8 PyQt5 passing
3.9	Qt5	Linux	3.9 PyQt5 passing

PyMoDAQ Public
forked from PyMoDAQ/PyMoDAQ

monkey-branch had recent pushes 4 minutes ago

Compare & pull request

monkey-branch 11 branches 46 tags Go to file Add file <> Code

This branch is 12 commits ahead, 1 commit behind PyMoDAQ:main. Contribute Sync fork

quantumm Add the monkey in the README. 24aa137 20 minutes ago 1,308 commits

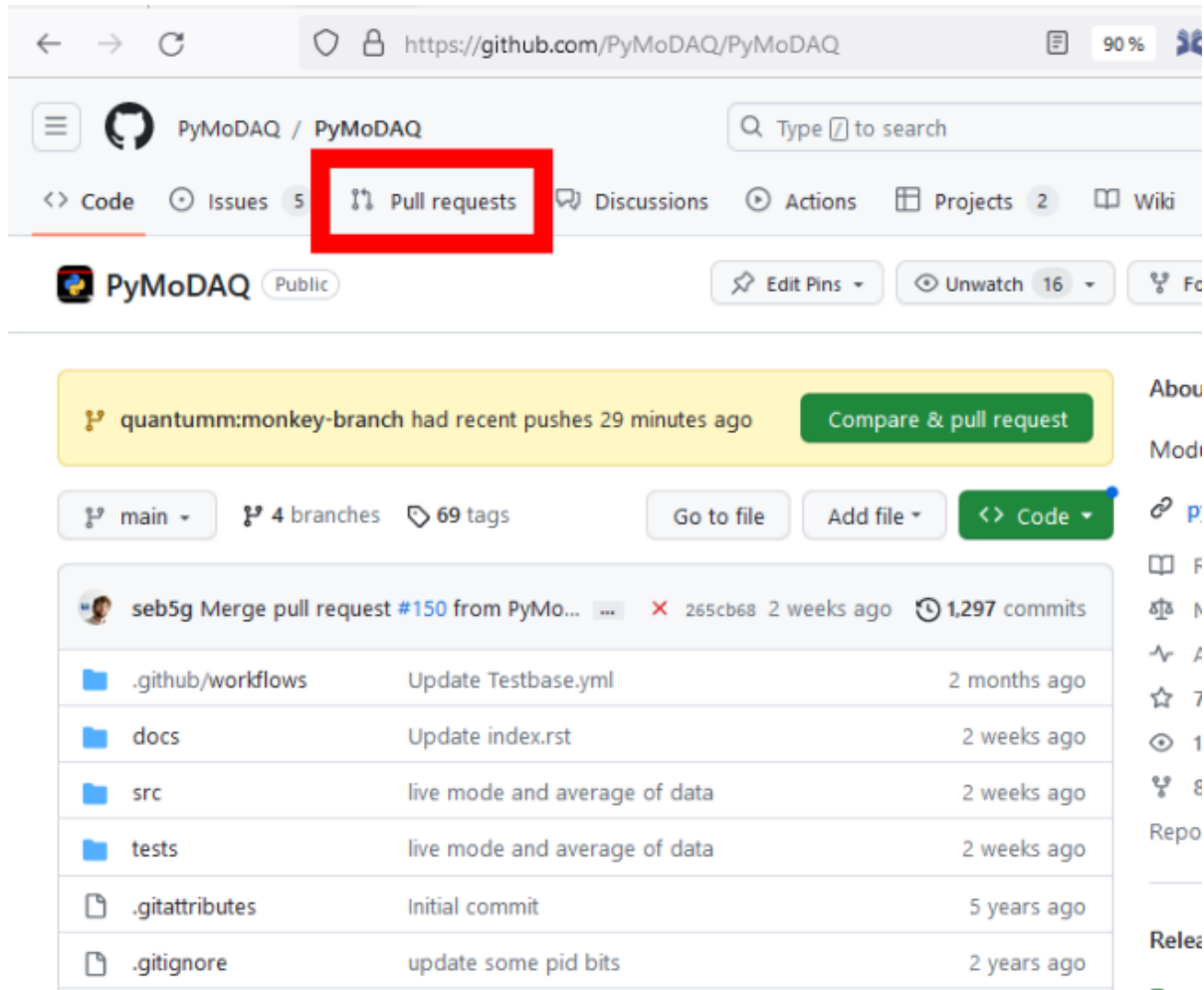
.github/workflows	Update Testbase.yml	2 months ago
docs	Update data_objects.rst	2 weeks ago
src	Update viewer.py	5 days ago
tests	axis slicing tests and slice with an integer	last week
.gitattributes	Initial commit	5 years ago

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).

upstream repository	branch	remote repository	branch
base repository: PyMoDAQ/PyMoDAQ	base: pymodaq-dev	head repository: quantumm/PyMoDAQ	compare: monkey-branch

✓ Able to merge. These branches can be automatically merged.



← → ↻ <https://github.com/PyMoDAQ/PyMoDAQ> 90 %

PyMoDAQ / PyMoDAQ

<> Code Issues 5 **Pull requests** Discussions Actions Projects 2 Wiki

PyMoDAQ Public Edit Pins Unwatch 16 Fork

quantumm:monkey-branch had recent pushes 29 minutes ago [Compare & pull request](#)

main 4 branches 69 tags Go to file Add file <> Code

seb5g Merge pull request #150 from PyMo... 265cb68 2 weeks ago 1,297 commits

.github/workflows	Update Testbase.yml	2 months ago
docs	Update index.rst	2 weeks ago
src	live mode and average of data	2 weeks ago
tests	live mode and average of data	2 weeks ago
.gitattributes	Initial commit	5 years ago
.gitignore	update some pid bits	2 years ago

Note that opening a PR does not prevent us from working on our remote repository anymore, while waiting for the answer of the owner of the upstream repository. If we continue to commit some changes to the branch that we used for our PR (the *monkey_branch* here), the PR will be automatically updated, and the new commits will be considered as part of the PR. If we want to pursue the work but not put the following commits in the PR, we can start a new branch from the *monkey_branch*.

7.5.3 How to create a new plugin/package for PyMoDAQ?

Author email	sebastien.weber@cemes.fr
Last update	january 2024
Difficulty	Intermediate

In this tutorial, we will learn how to create a brand new *plugin* either for adding instruments, models or extensions!

Prerequisite

We will suppose that you have followed these tutorials:

- *Basics of Git and GitHub*
- *How to modify existing PyMoDAQ's code?*

In the latter, we presented how to interact with existing repositories but what if:

- you have an instrument from a manufacturer that doesn't have yet its package!
- you want to build a brand new extension to the *DashBoard*!

No worries, you don't have to start from scratch, but from a fairly complete template package!

The PyMoDAQ's plugin template repository

Among all the PyMoDAQ related github repository, there is one that is not a real one. This is the `py-modaq_plugins_template` (see Fig. 7.104)

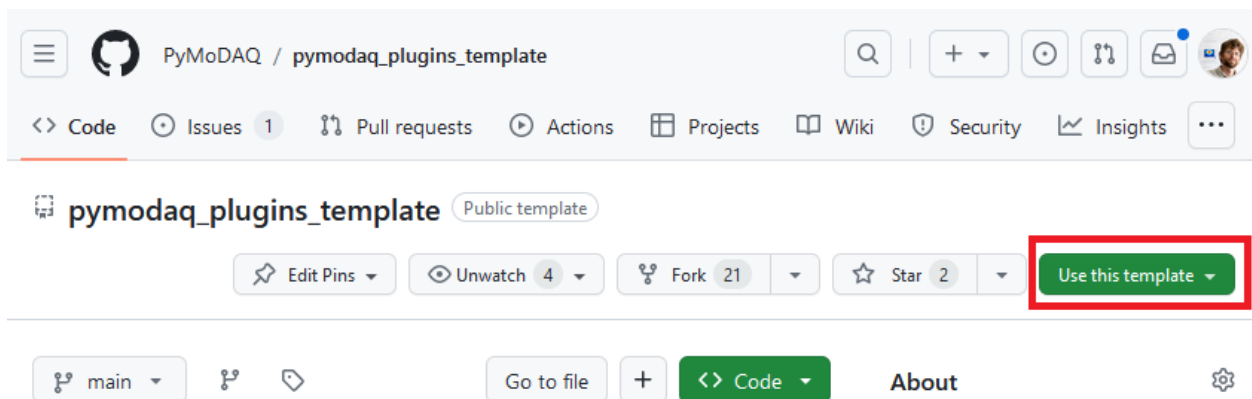


Fig. 7.104: The Template repository to create new plugin packages!

You see that on this repository home page, a new green button *Use this template* appeared (red box on figure). By clicking on it, you'll be prompted to *create a new repository*. In the next page, you'll be prompted to enter a owner and a name for the repo, see Fig. 7.105:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk ().*

Repository template


 PyMoDAQ/pymodaq_plugins_template ▾

Start your repository with a template repository's contents.

☐ **Include all branches**

Copy all branches from PyMoDAQ/pymodaq_plugins_template and not just the default branch.

Owner *

 PyMoDAQ ▾

Repository name *

/ pymodaq_plugins_myrepo

✔ pymodaq_plugins_myrepo is available.

Great repository names are short and memorable. Need inspiration? How about [verbose-octo-pancake](#) ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Grant your Marketplace apps access to this repository

PyMoDAQ is subscribed to 1 Marketplace app.



Travis CI *(auto-installed)*

Test and deploy with confidence



You are creating a public repository in the PyMoDAQ organization.

Create repository

Fig. 7.105: The creation page of the new plugin repository

In there, you can choose as a owner either yourself or the PyMoDAQ organisation if you're already part of it. If not but you are willing, just send an email to the mailing list asking for it and you'll be added and set as the manager of your future new plugin package. The name of the plugin as to follow the rule: `pymodaq_plugins_<my_repo_name>` where you have to replace `<my_repo_name>` by the name of the manufacturer if you're planning to add instruments or a clear name for your application/extension... Make it *Public* because we want to share our work within the PyMoDAQ community!

That's it, your new github repo compatible with PyMoDAQ is created. You now have to properly configure it!

Configuring a new plugin repository

For a correct configuration (for your plugin be installable and recognised by PyMoDAQ), you'll have to modify a few files and folders. [Fig. 7.106](#) highlight the package initial structure. You'll have to:

- rename with the new package name the two directories in highlighted red
- fill in the appropriate information in `plugin_info.toml` and `README.rst` files, highlighted in green
- rename the python instrument file, highlighted in purple with the dedicated instrument name (see [Story of an instrument plugin development](#) for details on instrument, python file and class name convention).
- add appropriate default settings in the `config_template.toml` file (do not rename it) in the resources folder,
- remove the unused instrument example files of the template repository in the `daq_move_plugins` and `daq_viewer_plugins` subfolders.
- Modify and configure the automatic publication of your package on the Pypi server (see [Publishing on Pypi](#))

Publishing on Pypi

In the Python ecosystem, we often install packages using the `pip` application. But what happens when we execute `pip install mypackage`? Well `pip` is actually looking on a web server for the existence of such a package, then download it and install it. This server is the Pypi [Python Package Index](#)

Developers who wish to share their package with others can therefore upload their package there as it is so easy to install it using `pip`. To do that you will need to create an account on Pypi:

Note: Until recently (late 2023) only a user name and password were needed to create the account and upload packages. Now the account creation requires double identification (can use an authentication app on your mobile or a token). The configuration of the Github action for automatic publication requires also modifications... See below.

You have to configure an API token with your pypi account. This token will allow you to create new package on your account, see [API Token](#) for more in depth explanation. This pypi package initial creation and later on subsequent versions upload may be directly triggered from Github using one of the configured Actions. An action will trigger some process execution on a distant server using the most recent code on your repository. The actions can be triggered on certain events. For instance, everytime a commit is made, an action is triggered that will run the tests suite and let developers know of possible issues. Another action is triggered when a *release* is created on github. This action will build the new version of the package (the released one) and upload the new version of the code on pypi. However your github account (at least the one that is the owner of the repository) should configure what Github call Secrets. Originally they were the pypi user name and password. Now they should be the `__token__` string as username and the API token generated on your pypi account as the password. The `yml` file corresponding to this action is called `python-publish.yml` stored in the `.github` folder at the root of your package. The content looks like this:

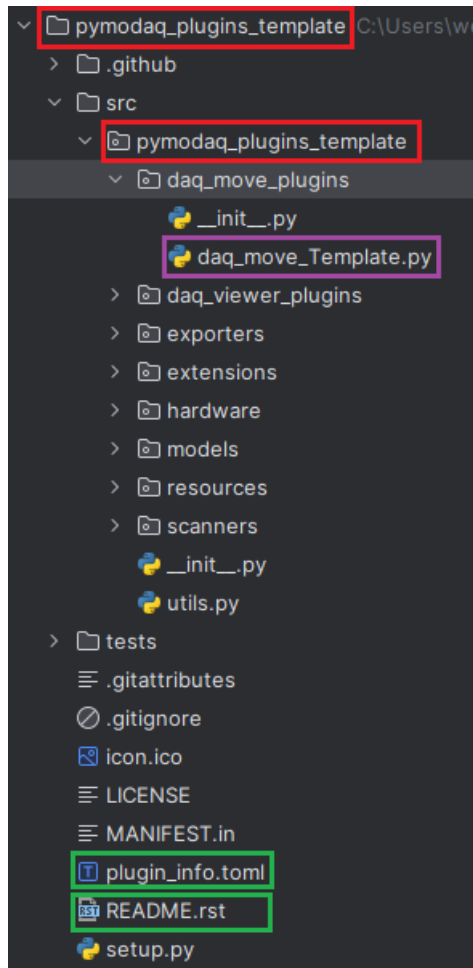


Fig. 7.106: The template package initial structure

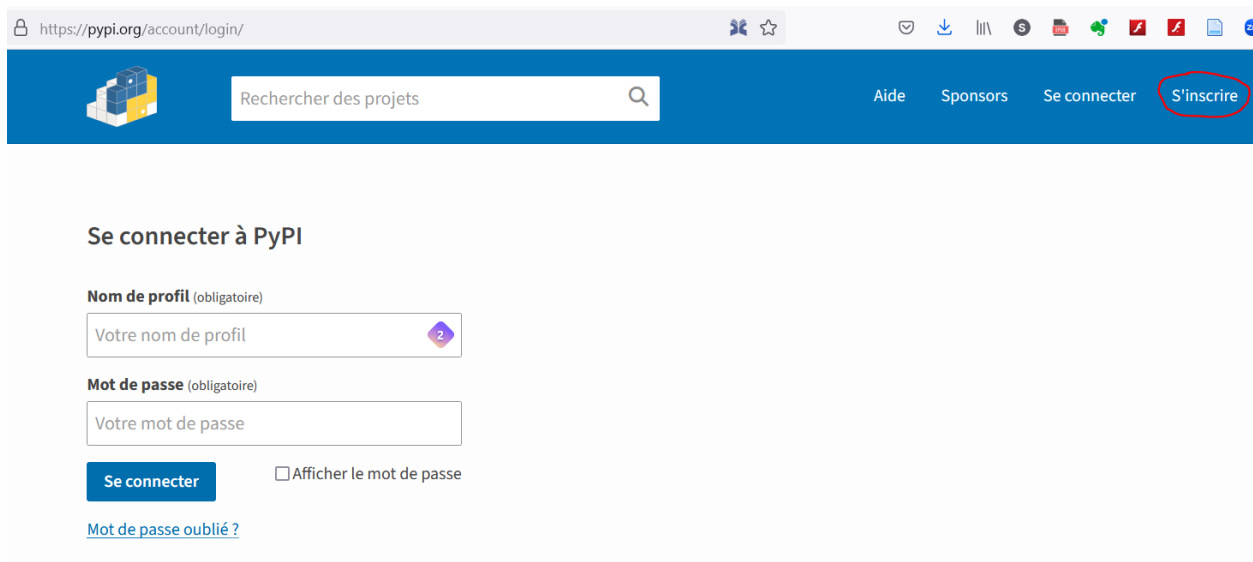


Fig. 7.107: Creation of an account on Pypi

```

name: Upload Python Package

on:
  release:
    types: [created]

jobs:
  deploy:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.11'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install setuptools wheel twine toml "pymodaq>=4.1.0" pyqt5

    - name: create local pymodaq folder and setting permissions
      run: |
        sudo mkdir /etc/.pymodaq
        sudo chmod uo+rw /etc/.pymodaq

    - name: Build and publish
      env:
        TWINE_USERNAME: ${ secrets.PYPI_USERNAME }
        TWINE_PASSWORD: ${ secrets.PYPI_PASSWORD }
      run: |
        python setup.py sdist bdist_wheel
        twine upload dist/*

```

were different jobs, steps and actions (*run*) are defined, like:

- execute all this on a ubuntu virtual machine (could be windows, macOS...)
- Set up Python: configure the virtual machine to use python 3.11
- Install dependencies: all the python packages necessary to build our package
- create local pymodaq folder and setting permissions: make sure pymodaq can work
- Build and publish: the actual thing we are interested in, building the application from the setup.py file and uploading it on pypi using the twine application

For this last step, some environment variable have been created from github secrets. Those are the `__token__` string and the API token. We therefore have to create those secrets on github. For this, you'll go in the *settings* tab (see Fig. 7.108) to create secrets either on the organization level or repository level (see PyMoDAQ example on the organisation level, Fig. 7.109).

That's it you should have a fully configured PyMoDAQ's plugin package!! You now just need to code your actual instrument or extension, for this look at *Story of an instrument plugin development*

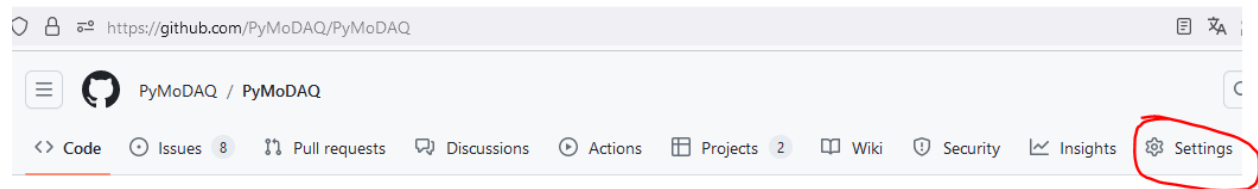


Fig. 7.108: Settings button on github

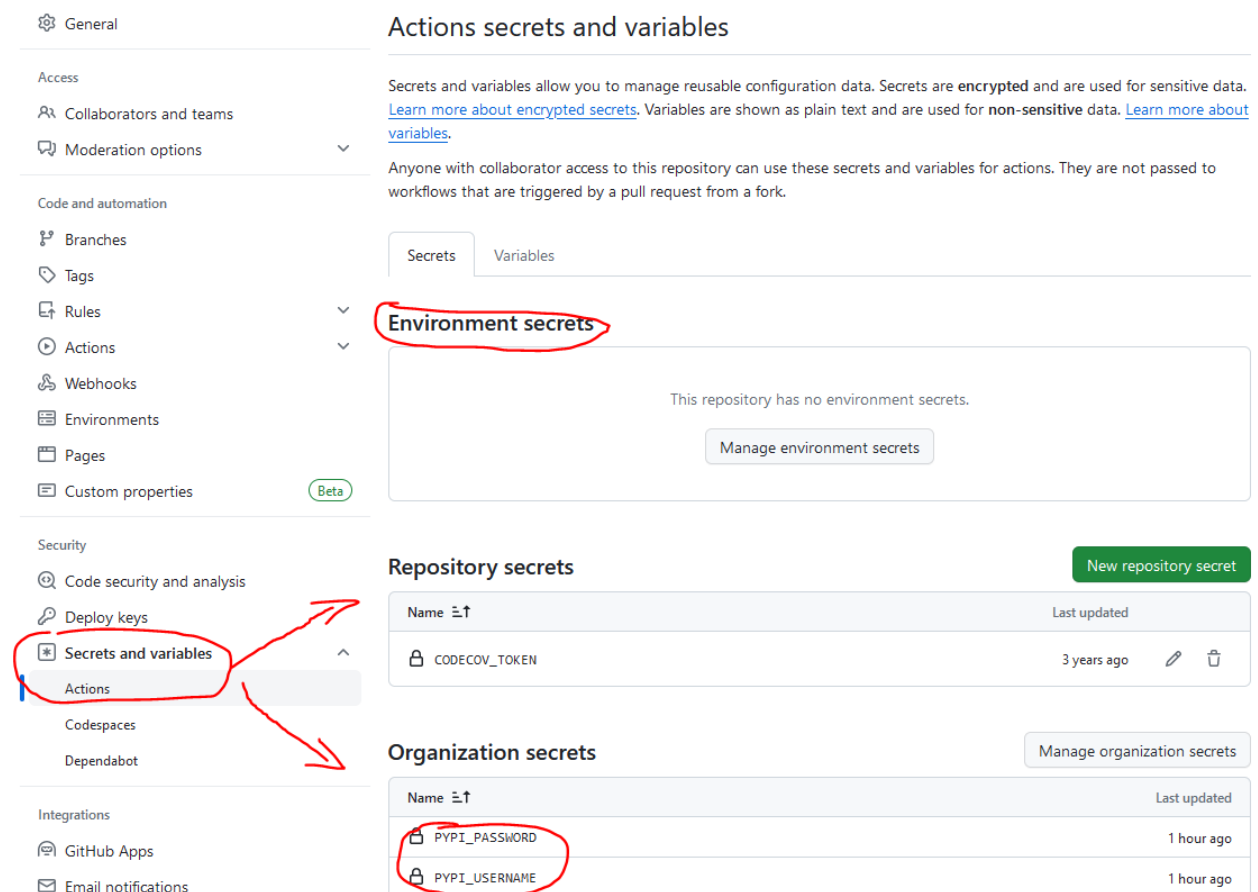


Fig. 7.109: Secrets creation on Github

Note: Starting with PyMoDAQ version 4.1.0 onwards, old github actions for publication and suite testing should be updated in the plugin packages. So if you are a package maintainer, please do so using the files from the template repository.

7.5.4 Story of an instrument plugin development

In this tutorial, we will explain step by step the way to develop an *instrument plugin*. It is a specific type of *plugin*, that will allow you to control your device through PyMoDAQ.

As PyMoDAQ is not a library for professional developers, we consider that you reader do not know anything about how the development of an open source project works. We will take the time to start from scratch, and allow us to expand the scope of this documentation out of PyMoDAQ itself, to introduce Python environment, Git, external python libraries...

Rather than looking for a general and exhaustive documentation, we will illustrate the development flow with a precise example. We will go step by step through the development of the **PI E-870 controller** instrument plugin: from the reception of the device up to controlling it with PyMoDAQ. This one is chosen to be quite simple and standard. This controller can be used for example to control PiezoMike actuators, as illustrated below.

PI E-870 controller



PiezoMike actuators on an optical mount



Fig. 7.110: The PI E-870 controller and PiezoMike actuators mounted on an optical mount.

The benefit of writing an instrument plugin is twofold:

- Firstly, you will learn a lot about coding, and coding the good way! Using the most efficient tools that are used by professional developers. We will introduce how to use Python editors, linters, code-versioning, testing, publication, bug reporting, how to integrate open-source external libraries in your code... so that in the end you have a good understanding of how to develop a code in a collaborative way.
- Secondly, writing an instrument plugin is a great opportunity to dig into the understanding of your hardware: what are the physical principles that make my instrument working? How to fully benefit of all its functionalities? What are the different communication layers between my device and my Python script?

Writing an instrument plugin is very instructive, and perfectly matches a student project.

The controller manual

Let's not be too impatient and take the time to read the [controller manual](#), in the introduction we can read

“the E-870 is intended for open-loop operation of PIShift piezo inertia drives.” (page 3)

Ok but what is this *PIShift* thing? It is quite easy to find those videos that make you understand in a few tens of seconds the operating principle of the actuator:

[PIShift drive principle](#)

[PiezoMike linear actuator](#)

Nice! :)

What is *open-loop* operation? It means the system has no reading of the actuator position, as opposed to a *close-loop* operation. The open-loop operation is simpler and cheaper, because it does not require any *encoder* or *limit switch*, but it means that you will have no absolute reference of your axis, and less precision. This is an important choice when you buy an actuator, and it depends on your application. This will have big impact on our instrument plugin development.

“The E-870 supports one PIShift channel. The piezo voltage of the PIShift channel can be transferred to one of two (E-870.2G) or four (E-870.4G) demultiplexer channels, depending on the model. Up to two or four PIShift drives can be controlled serially in this manner.” (page 19)

Here we learn that in this controller, there is actually only one *channel* followed by a demultiplexer that will distribute the amplified current to the addressed axis. This means that only one axis can be moved at a time, the drives can only be controlled *serially*. This also depends on your hardware, and is an important information for the instrument plugin development.

The installer

It is important to notice that **PyMoDAQ itself does not necessarily provide all the software needed to control your device**. Most of the time, you have to install *drivers*, which are pieces of software, specific to each device, that are indispensable to establish the communication between your device and the operating system. Those are necessarily provided by the manufacturer. The ones you will install can depend on your operating system, and also on the way you establish the communication between them. Most of the time, you will install the USB driver for example, but this is probably useless if you communicate through Ethernet.

Let's now run the *installer* provided in the CD that comes with the controller. The filename is *PI_E-870.CD_Setup.exe*. It is an *executable* file, which means that it hosts a program.

Fig. 7.111: The GUI of the installer.

On the capture on the right, you can see what it will install on your local computer, in particular:

- Documentation.
- A *graphical user interface* (GUI) to control the instrument, called the *PI E870Control*.
- Labview drivers: we will NOT need that! ;)
- A DLL library: PI GCS DLL. We will talk about that below.
- Some programming examples to illustrate how to communicate with the instrument depending on the programming language you use.
- USB drivers.

Whatever the way you want to communicate with your device, you will need the drivers. Thus, again, **you need to install them before using PyMoDAQ**.

Once those are installed, plug the controller with a USB cable, and go to the *Device settings* of Windows. An icon should appear like in the following figure. It is the first thing to check when you are not sure about the communication with your device. If this icon does not appear or there is a warning sign, change the cable or reinstall the drivers, no need to go further. You can also get some information about the driver.

Fig. 7.112: The *Device settings* window on Windows.

In the following, we will follow different routes, as illustrated in the following figure to progressively achieve the complete control of our actuator with PyMoDAQ. In the following we will name them after the color on the figure.

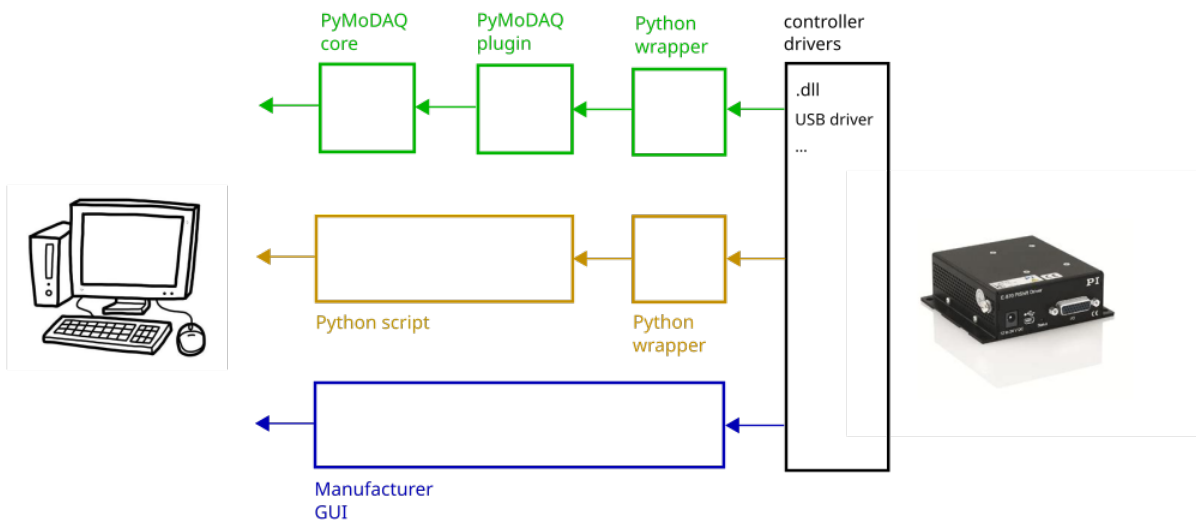


Fig. 7.113: The different routes (blue, gold, green) to establish the communication between the computer and the controller.

The blue route: use the manufacturer GUI

The simplest way to control your device is to use the GUI software that is provided by the manufacturer. It is useful while you are under development, but will be useless once you have developed your plugin. PyMoDAQ will replace it, and even provide much broader functionalities. While a specific manufacturer GUI talks to only one specific device, PyMoDAQ provides to you a common framework to talk to many different instruments, synchronize them, save the acquisitions, and many more!

In the main tab, we found the buttons to send relative move orders, change the number of steps, change the controlled axis (in this example we can control 4 axis). **Check all that works properly.**

The second tab goes to a lower level. It allows us to directly send commands from the PI GCS library. We will see that below.

Whenever you want to control a device with PyMoDAQ for the first time, even if you do not develop a plugin, **you should first check that the manufacturer software is able to control your device.** It is a prerequisite before using

Fig. 7.114: Captures of the GUI provided by PI. **Left:** Interface to move the actuators and change the axis. **Right:** Interface to send GCS commands (see below).

PyMoDAQ. By doing so we already checked a lot of things:

- The drivers are correctly installed.
- The communication with the controller is OK.
- The actuators are moving properly.

We are now ready for the serious part!

A shortcut through an existing green route? Readily available PyMoDAQ instrument plugins

Before dedicating hours of work to develop your own solution, we should check what has already been done. If we are lucky, some good fellow would already have developed the instrument plugin for our controller!

Here is the [list of readily available plugins](#).

Each plugin is a *Python package*, and also a *Git repository* (we will talk about that later).

By convention, an instrument plugin can be used to control several devices, **but only if they are from the same manufacturer**. Those several hardwares can be actuators or detectors of different dimensionalities. The **naming convention for an instrument plugin** is

pymodaq-plugins-<manufacturer name>

Note: Notice the “s” at the end of “plugins”.

Note: Any kind of plugin should follow the naming convention *pymodaq-plugins-<something more specific>*, but an instrument plugin is a specific kind of *plugin*. For (an advanced) example, imagine that we create a beam pointing stabilization plugin, and that this system uses devices from different companies. We could have an actuator class that controls a SmarAct optical mount, a detector class that control a Thorlabs camera, and a *PID model* specifically designed for our needs. In that case we could use the name *pymodaq-plugins-beam-stabilization*.

All the plugins that are listed there can directly be installed with the *plugin manager*.

Some of those - let say the *official* ones - are hosted by the [PyMoDAQ organization on GitHub](#), but they can also be hosted by other organizations. For example, the repository [pymodaq-plugins-greateyes](#) is hosted by the ATTOLab organization, but you can directly install it with the plugin manager.

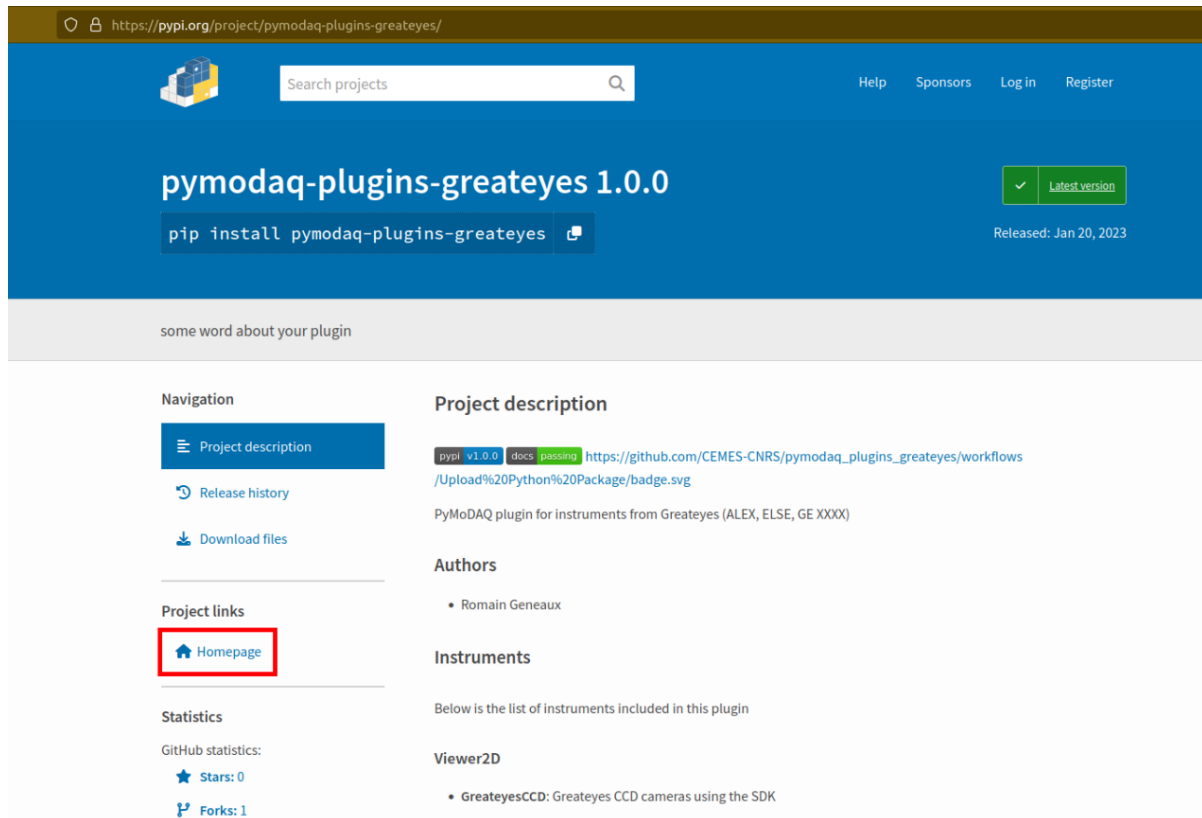
Remember that the already developed plugins will give you a lot of working examples, probably the way you will develop your own plugin will be very similar to one that already exist.

It sounds like we are very lucky... the PI plugin already exists!

Let's try it!

Firstly, we have to *install PyMoDAQ* in a dedicated Python environment, that we will call *pmd_dev* in this tutorial.

Now that PyMoDAQ is installed and you have activated your environment (the lign of your terminal should start with (*pmd_dev*)), we will try to install the PIinstrument plugin with the plugin manager. In your terminal, execute the following command



Navigation

- Project description
- Release history
- Download files

Project links

- Homepage**

Statistics

GitHub statistics:

- Stars: 0
- Forks: 1

Project description

`pypi v1.0.0` `docs passing` https://github.com/CEMES-CNRS/pymodaq_plugins_greateyes/workflows/Upload%20Python%20Package/badge.svg

PyMoDAQ plugin for instruments from Greateyes (ALEX, ELSE, GE XXXX)

Authors

- Romain Geneaux

Instruments

Below is the list of instruments included in this plugin

Viewer2D

- GreateyesCCD**: Greateyes CCD cameras using the SDK

Fig. 7.115: The PyPI page of the greateyes plugin. If you click on *Homepage* you will find the Git repository page.

Physik Instrumente Stages	<ul style="list-style-type: none"> Sebastien J. Weber 	1.0.0	<p>Actuators</p> <ul style="list-style-type: none"> PI: All stages compatible with the GCS2 library. Tested on E-816, C-863 (mercury DC/Stepper), C-663, E-545 PI_MMC: old controller and stages using the 32 bits MMC dll (requires 32bit python) C-862 controller
---------------------------	--	-------	---

Fig. 7.116: There is already a PI plugin in the list of available plugins.

```
(pmd_dev) >plugin_manager
```

This will pop-up a window like this, select the plugin we are interested in and click *Install*

Fig. 7.117: Interface of the plugin manager.

Now let's launch a DAQ_Move

```
(pmd_dev) >daq_move
```

Fig. 7.118: DAQ Move interface.

- (1) The list of available actuator contains the *PI* one, that sounds good!
- (2) Let select the *USB* connection type.
- (3) The list of available devices contains our controller with his serial number! That sounds really good because it means that the program can see the controller!
- (4) Let's launch the initialization! Damn. The LED stays red! Something went wrong...

In a perfect world this should work, since we followed the proper way. But PyMoDAQ is a project under development, and some bugs may appear. Let's not be discouraged! Actually we should be happy to have found this bug, otherwise we would not have the opportunity to explain how to face it.

What do we do now?

First, let's try to get more information about this bug. PyMoDAQ automatically feeds a log file, let's see what it has to tell us. You can find it on your computer at the location

<OS username>/pymodaq/log/pymodaq.log

or you can open it through the Dashboard menu :

File > Show log file

It looks like this

Fig. 7.119: The log file of PyMoDAQ after trying to initialize the plugin.

This log file contains a lot of information that is written during the execution of PyMoDAQ. It is sorted in chronological order. If you find a bug, the first thing to do is thus to go at the end of this file.

In the above capture, we see that the first line indicates the moment we clicked on the *Initialization* button of the interface.

In the following we see that an error appeared: **Unknown command (2)**. The least we can say is that it is not crystal clear to deduce the error from this!

At this point, we will not escape from digging into the code. If you do not feel like it, there is a last but very important thing that you can do, which is to **report the bug**. Try to detail as much as possible every step of your problem, and copy paste the part of the log file that is important. Even if you do not provide any solution, this reporting will be a usefull step to make PyMoDAQ better.

You dispose of several ways to do so.

- (1) Leave a message in the PyMoDAQ mailing list pymodaq@services.cnrs.fr.
- (2) Leave a message to the developer of the plugin.
- (3) Raise an issue on the GitHub repository associated to the plugin (you need to create an account, which is free). This last option is the most efficient because it targets precisely the code that raises a problem. Plus it will stay archived and visible to anyone that would face the same problem in the future.

Fig. 7.120: How to raise an issue on a GitHub repository.

Now we have gone as far as possible we could go without digging into the code, but if you are keen on it, let's continue on the gold route (Fig. 7.113)!

The gold route: control your device with a Python script

We are now ready to tackle the core of this tutorial, and learn how to write a Python code to move our actuator. Let's first introduce some important concepts.

What is a DLL?

As you may have noticed, the installer saved locally a file called *PI_GCS2_DLL.dll*.

The .dll file is a *library* that contains functions that are written in C++. It is an [API](#) between the controller and a computer application like PyMoDAQ or the PI GUI. It is made so that the person that intends to communicate with the controller is forced to do it the proper way (defined by the manufacturer's developers). You cannot see the content of this file, but **it is always provided with a documentation**.

If you want to know more about DLLs, have a look at the [Microsoft documentation](#).

Note: We suppose in this documentation that you use a Windows operating system, because it is the vast majority of the cases, but PyMoDAQ is also compatible with Linux operating systems. If you wish to control a device with a Linux system, you have to be careful during your purchase that your manufacturer provides Linux drivers, which is unfortunately not always the case. The equivalent of the .dll format for a Linux operating system is a .so file. PI provide such file, which is great! The development of Linux-compatible plugins will be the topic of another tutorial.

The whole thing of the gold route is to find how to talk to the DLL through Python.

In our example, PI developed a DLL library that is common to a lot of its controllers, called the *GCS 2.0 library* (it is the 2.0 version that is adapted to our controller). The [associated documentation](#) is quite impressive at first sight: 100+ (harsh!) pages.

This documentation is supposed to be exhaustive about all the functions that are provided by the library to communicate with a lot of controllers from PI. Fortunately, we will only need very few of them. The challenge here is to pick up the good information there. This is probably the most difficult part of an instrument plugin development. This is mostly due to the fact that there is no standardization of the way the library is written. Thus the route we will follow here will probably not be exactly the same for another device. Here we also depend a lot on the quality of the work of the developers of the library. If the documentation is shitty, that could be a nightmare.

Note: Our example deals with a C++ DLL, but there are other ways to communicate with a device: ASCII commands, .NET libraries (using [pythonnet](#))...

What is a Python wrapper?

As we have said in the previous section, the DLL is written in C++. We thus have a problem because we talk the Python! A *Python wrapper* is a library that defines Python functions to call the DLL.

PIPython wrapper

Now that we introduced the concepts of DLL and Python wrapper, let's continue with the same philosophy. We want to be efficient. We want to go straight to the point and code as little as possible. We are probably not the first ones to want to control our PI actuator with a Python script! Asking a search engine about “*physik instrumente python*”, we directly end up to the PI Python wrapper called *PIPython*.

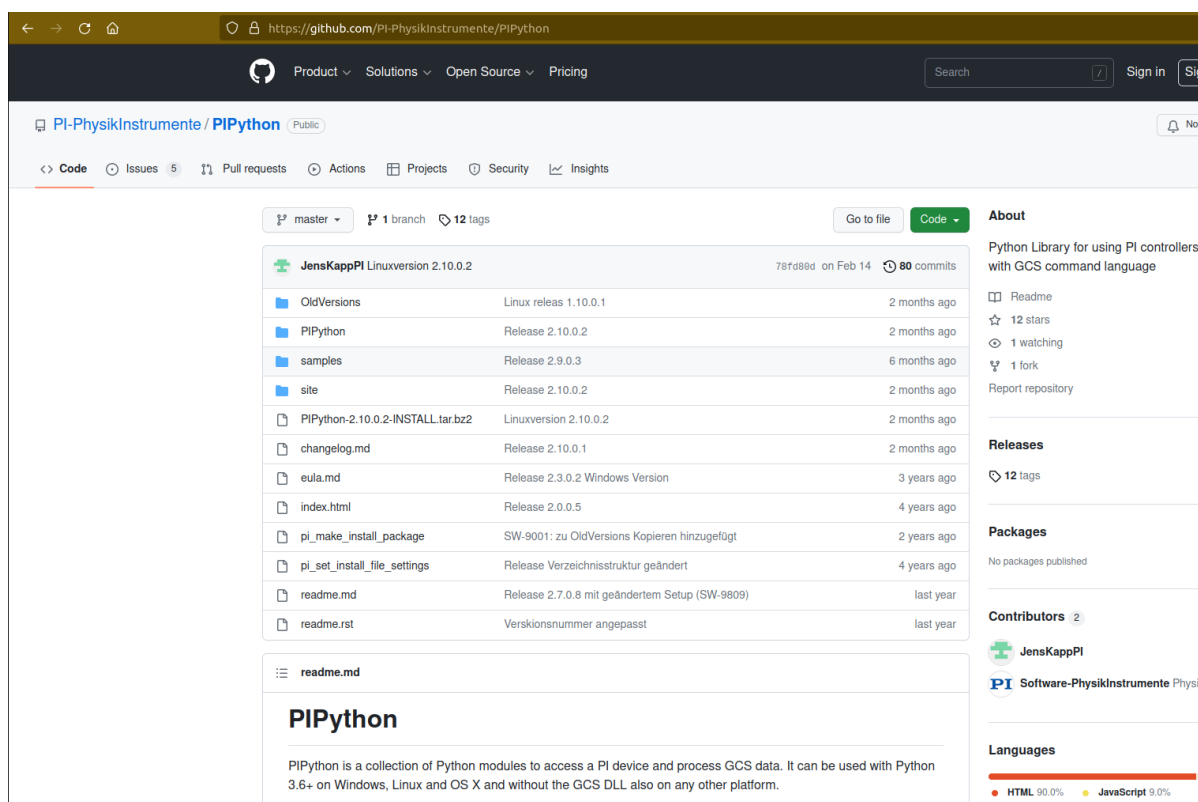


Fig. 7.121: The PIPython repository on GitHub.

We can now understand a bit better the error given in the PyMoDAQ log earlier. It actually refers to the *pipython* package. This is because the PI plugin that we tested actually uses this library.

Note: All the Python packages of your environment are stored in the *site-packages* folder. In our case the complete path is `C:\Users\<OS_username>\Anaconda3\envs\pmd_dev\Lib\site-packages`. Be careful to not end up in the *base* environment of Anaconda, which is located at `C:\Users\<OS_username>\Anaconda3\Lib\site-packages`.

That's great news! The PI developers did a great job, and this will save us a lot of time. Unfortunately, this is not always the case. There are still some less serious suppliers that do not provide an open-source Python wrapper. You should consider this as a serious argument *before* you buy your lab equipment, as it can save you a lot of time and

struggle. Doing so, you will put some pressure on the suppliers to develop Python open-source code, so that we can free our lab instruments!

External open-source libraries

In our example, our supplier is serious. Probably the wrapper it developed will do a good job. But let us imagine that it is not the case, and take a bit of time to present a few *external libraries*.

PyMoDAQ is of course not the only project of its kind. You can find on the internet a lot of non-official resources to help you communicate with your equipment. Some of them are so great and cover so much instruments that you should automatically check if your device is supported. Even if your supplier proposes a solution, it can be inspiring to have a look at them. Let's present the most important ones.

PyLabLib

[PyLabLib](#) is a very impressive library that interfaces the most common instruments that you will find in a lab:

- Cameras: Andor, Basler, Thorlabs, PCO...
- Stages: Attocube, Newport, SmarAct...
- Sensors: Ophir, Pfeiffer, Leybold...

... but also lasers, scopes, Arduino... to cite a few!

Here is the [complete list of supported instruments](#).

Here is the [GitHub repository](#).

PyLabLib is extremely well documented and the drivers it provides are of extremely good quality: a reference!

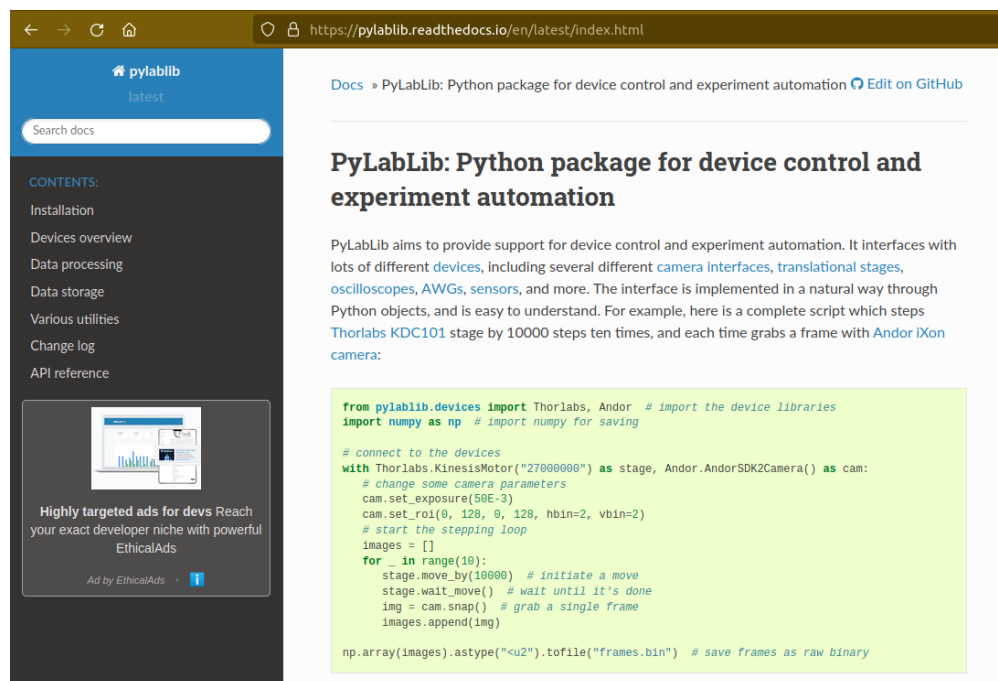


Fig. 7.122: The PyLabLib website.

Of particular interest are the **camera drivers**, that are often the most difficult ones to develop. It also proposes a GUI as a side project to control cameras: [cam control](#).

Instrumental

[Instrumental](#) is also a very good library that you should know about, which covers different instruments.

Here is the [list of supported instruments](#).

As you can see with the little script that is given as an example, it is super easy to use.

Instrumental is particularly good to create drivers from DLL written from C where one have the header file, autoprocessing the function signatures...

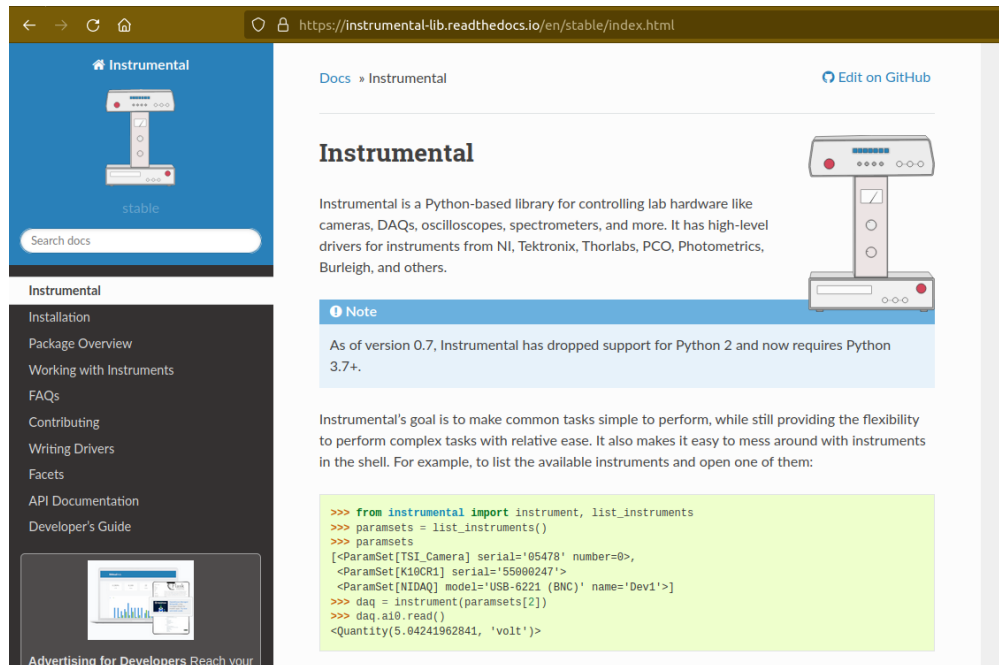


Fig. 7.123: The Instrumental website.

PyMeasure

[PyMeasure](#) will be our final example.

You can find here the [list of supported instruments](#) by the library.

This library is very efficient for all instruments that communicate through ASCII commands ([pyvisa](#) basically) and makes drivers very easy to use and develop.

Installation of external libraries

The installation of those libraries in our environment cannot be simpler:

```
(pmd_dev) >pip install <library name>
```

This list is of course not exhaustive. Those external resources are precious, they will often provide a good solution to start with!

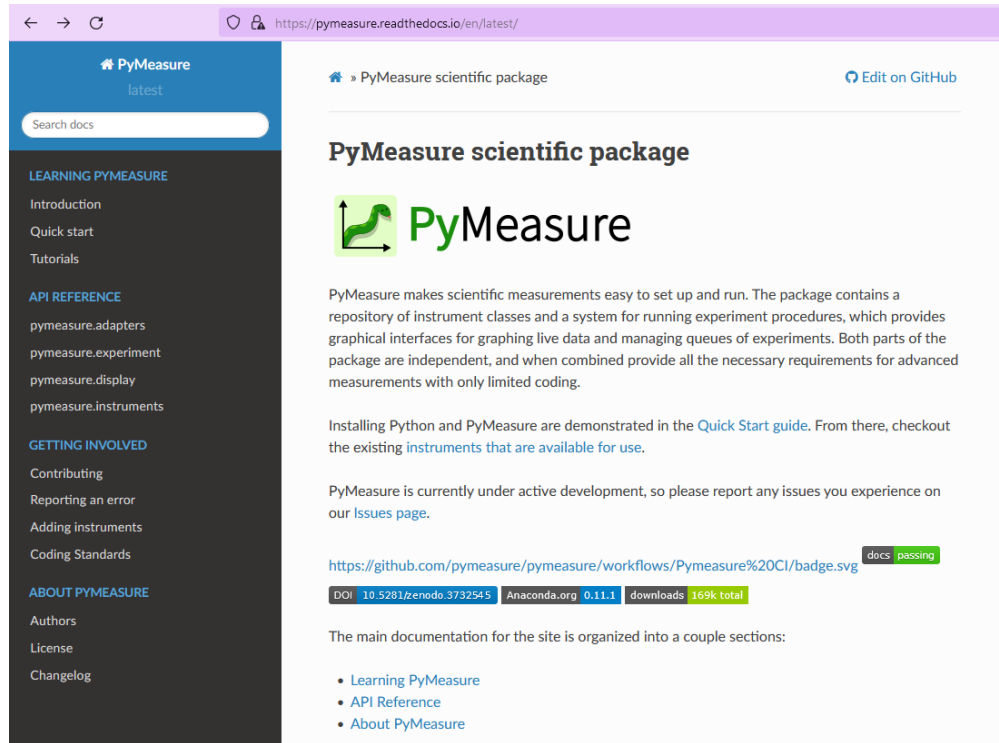


Fig. 7.124: The PyMeasure website.

Back to PIPython wrapper

Let's now go back to our E870 controller, it is time to test the PIPython wrapper!

We will install the package *pipython* in our *pmd_dev* environment

```
(pmd_dev) >pip install pipython
```

after the installation, we can check that the dependencies of this package have been installed properly using

```
(pmd_dev) >conda list
```

which will list all the packages that are installed in our environment

Here we found the [documentation of the wrapper](#).

It proposes a very simple script to establish the communication. Let's try that!

We will use the [Spyder](#) IDE to deal with such simple script, which is freely available. If you already installed an Anaconda distribution, it should already be installed.

Let's open it and create a new file that we call *script_pmd_pi_plugin.py* and copy-paste the script.

It is important that you configure Spyder properly so that the *import* statement at the beginning of the file will be done in our Python environment, where we installed the PIPython package. For this, click on the *settings* icon as indicated in the following capture.

The following window will appear. Go to the *Python interpreter* tab and select the Python interpreter (a *python.exe* file for Windows) which is at the root of your environment (in our case our environment is called *pmd_dev*. Notice that it is located in the *envs* subfolder of Anaconda). Do not forget to *Apply* the changes.

Let's now launch the script clicking the *Run* button. A pop-up window appears. We have to select our controller, which is uniquely identify by its serial number (SN). In our exemple it is the one that is underlined in blue in the capture. It

```
python 3.8.13 h6244533_0
python-dateutil 2.8.2 pypi_0 pypi
python-markdown-math 0.8 pypi_0 pypi
pythonnet 3.0.0.post1 pypi_0 pypi
pytz 2022.4 pypi_0 pypi
pyusb 1.2.1 pypi_0 pypi
pyvisa 1.13.0 pypi_0 pypi
pywin32 304 pypi_0 pypi
pyzmq 25.0.0 pypi_0 pypi
qdarkstyle 3.1 pypi_0 pypi
qtpy 2.2.1 pypi_0 pypi
readme-renderer 37.2 pypi_0 pypi
requests 2.28.1 pypi_0 pypi
retext 8.0.0 pypi_0 pypi
scipy 1.9.1 pypi_0 pypi
setuptools 63.4.1 py38haa95532_0
simple-pid 1.0.1 pypi_0 pypi
six 1.16.0 pypi_0 pypi
spyder-kernels 2.1.3 pypi_0 pypi
sqlite 3.39.3 h2bbff1b_0
stack-data 0.6.2 pypi_0 pypi
tabledata 1.3.0 pypi_0 pypi
tables 3.6.1 pypi_0 pypi
tcolorpy 0.1.2 pypi_0 pypi
```

Fig. 7.125: List (partial) of the packages that are installed in our environment after installing *pipython*. We can check that the packages *pyusb*, *pysocket* and *pyserial* are there, as requested by the documentation.

The screenshot shows the PIPython Quick Start documentation page. The sidebar on the left contains a navigation menu with links to Home, Quick Start, Requirements, Establishing communication, Arguments, Setter functions, Getter functions, Return values, Some useful information, Helper functions, Debug logging, GCSError and error check, Big data, Textual interface, Device Connection, Data Recorder GCS 2.0, Data Recorder GCS 3.0, History, and EULA. The main content area is titled 'Quick Start' and includes a 'Requirements' section with a list of packages to install: PyUSB, PySocket, and PySerial. Below this is an 'Establishing communication' section that describes the GCSDevice class and provides a code snippet for connecting to a device.

```
from pipython import GCSDevice
pdevice = GCSDevice('C-884')
pdevice.InterfaceSetupDlg()
print(pdevice.qIDN())
pdevice.CloseConnection()
```

Fig. 7.126: *Quick Start* documentation of PIPython to establish the communication with a controller.

Fig. 7.127: Running the PIPython *quickstart* script in the Spyder IDE.

Fig. 7.128: Configure the good Python interpreter in Spyder.

seems like nothing much happens...

Fig. 7.129: Communication established!

...but actually, **we just received an answer from our controller!**

The script returns the reference and the serial number of our controller. Plus, we can see in the *Variable explorer* tab that the *pidevice* variable is now a Python object that represents the controller. For now nothing happens, but this means that our system is ready to receive orders. This is a big step!



Fig. 7.130: System ready.

Now, we have to understand how to play with this *GCSDDevice* object, and then we will be able to play with our actuators! First, we will blindly follow the *quickstart* instructions of PIPython, and try this script

Fig. 7.131: Script suggested by the *quickstart* instructions of PIPython. In our case it returns an error.

Note: If at some point you lose the connection with your controller, e.g. you cannot see its SN in the list, do not hesitate to reset the Python kernel. It is probably that the communication has not been closed properly during the last execution of the script.

Unfortunately this script is not working, and returns *GCSError: Unknown command (2)*.

RRRRRRRRRRRRrrrrrrrrrrrr!! Ok... this is again a bit frustrating. Something should be quite not precise in the documentation, so we [raised an issue](#) in the GitHub repository to explain our problem.

Anyway, that gives us the opportunity to dig into the DLL library!

The first part of the error message indicates that this error is raised by the GCS library. If we search *Unknown command* in the DLL manual, we actually found it

Fig. 7.132: GCS documentation page 109.

This is actually the error number 2, that explains the (2) at the end of the error message. Unfortunately, the description of the error does not help us at all. Still, it is categorized as a *controller error*. Plus, the introduction of the section remind us that the PI GCS is a library that is valid for a lot of controllers that are sold by the company. Then, we should expect that some commands of the library cannot be used with any controller. This is also confirmed elsewhere in the documentation.

Fig. 7.133: GCS documentation page 29.

Ok, it is more clear now, our controller is telling us that he does not know the *MOV* command! But **how can we know the commands that are valid for our controller?** Here again we will find the answer in the GCS manual (the E870 controller manual is not of great help, but the [E872 manual](#) also gives the list of available commands).

At first, this manual looks very difficult to diggest. But actually most of it is dedicated to precise definition of each of the command, and this will be needed only if we actually use it. One should notice that some are classified as *communication functions*. They are used to establish the communication with the controller, depending on the *communication protocol* that is used (RS232, USB, TCP/IP...). But this is not our problem right now.

Let's look at the *functions for GCS commands*. There is a big table that summarizes all the functions with a short description. We should concentrate on that. Here we understand that actually most of those functions can for sure not be used with our controller. As we have seen earlier in this tutorial, our controller is made for *open-loop* operation. Thus, we can already eliminate all the functions mentioning "close-loop", "referencing", "current position", "limit", "home", "absolute"... but on the contrary all the descriptions mentioning "relative", "open-loop" should trigger our attention. Notice that some of them start with a *q* to inform that they are *query* functions. They correspond to GCS commands that terminate with a question mark. They ask the controller for an information but do not send order. They are thus quite safe, since they will not move a motor for example. Within all those we notice in particular the *OSM* one, which seems a good candidate to make a relative move

Fig. 7.134: GCS OSM command short description, page 22.

and the *qHLP* one, that seems to answer our previous question!

Let's try that! Here is what the controller will answer

That's great, we now have the complete list of the commands that are supported by our controller. Plus, within it is the *OSM* one, that we noticed just before!

Let's now look at the detailed documentation about this command

It seems quite clear that it takes two arguments, the first one seems to refer to the axis we want to move, and the second one, non ambiguously, refers to the number of steps we want to move. So let's try the following script (if you are actually testing with a PiezoMike actuator **be careful that it is free to move!**)

It works! We did it! We managed to move our actuator with a Python script! Yeaaaaaaaah! :D

Ok let just tests the other axis, we modify the previous script with a 2 as the first parameter of the command

Another error... Erf! That was too easy apparently!

Here, the DLL documentation will not be of great help. It is not clear what is the difference between an *axis* and a *channel*. We rather have to remember what we learnt from the controller manual at the begining of this tutorial. The E870 has actually only one *channel* that is followed by a demultiplexer. So actually, what we have to do, when we want

Fig. 7.135: GCS qHLP command short description, page 24.

Fig. 7.136: E870 answer to the qHLP command.

to control another axis, is to change the configuration of the demultiplexer, which is explained in the *Demultiplexing* section of the manual. Here are described the proper GCS commands to change the axis.

Let's translate that into a Python script

After running again the script with the OSM command, we actually command the second axis! :D

This is the end of the gold route! That was the most difficult part of the tutorial. Because there is no global standard about how to write a DLL library, it is always a bit different depending on the device you want to control. We are in this route very dependent on the quality of the work of the developpers of our supplier, especially on the documentation. Thus, it is always a bit of an investigation throughout all the documentations and the libraries available on the internet.

All this work has been the opportunity for us to understand in great details the working principles of our device, and to get a *complete* mastering of all its functionalities. We now master the basics to order anything that is authorized by the GCS library to our controller through Python scripts!

If at some point you are struggling too much in this route, do not hesitate to ask for help. And if you find some bugs, do not hesitate to post an issue. Those are little individual steps that make an open source project works, they are very important!

I've found nothing to control my device with Python! :(

In the example of this tutorial, our supplier did a good job and provides a good Python wrapper. It was then relatively simple.

If in your case, after a thorough investigation of your supplier website and external libraries you found no resource, it is time to take your phone and call your supplier. He may have a solution for you. If he refuses to help you, then you will have to write the Python wrapper by your own. It is a piece of work, but doable!

First, you will need the DLL documentation and the .dll file.

Then, one problem you will have to face is that the Python types are different from C, the language that is used in the DLL. You thus have to make more rigorous type declarations that you would do with Python. Hopefully, the `cypes` library is here to help you! The PIPython wrapper itself uses this library (for example see: `pipython/pidevice/interfaces/gcsdll.py`).

Finally, found examples of codes that are the closest possible to your problem. You can look for examples in other instrument plugins, the wrappers should be in the *hardware* subfolder of the plugin:

- [SmarAct MCS2 wrapper](#)
- [Thorlabs TLPM wrapper](#)

Fig. 7.137: GCS OSM command detailed description.

Fig. 7.138: Script using the OSM command to move the actuator.

Fig. 7.139: First test of a script using the OSM command to move the second axis of the controller.

The green route: control your device with PyMoDAQ

Now that we know how to control our actuators with Python, it will be quite simple to write our PyMoDAQ plugin, that is what we will learn in this section!

Before doing so, we have to introduce a few tools and prepare a few things that are indispensable to work properly in an open-source project.

What is GitHub?

You probably noticed that we refer quite a lot to this website in this tutorial, so what it is exactly?

GitHub is basically a website that provides services to store and develop open-source projects. Very famous open-source projects are stored on GitHub, like the [Linux kernel](#) or the software that runs [Wikipedia](#). PyMoDAQ is also stored on GitHub.

It is based on *Git* that is currently the most popular *version control software*. It is made to keep track of every modification that has been made in a folder, and to allow multiple persons to work on the same project. It is a very powerful tool. If you do not know about it, we recommend you to make a few research to understand the basic concepts. In the following, we will present a concrete example about how to use it.

The following preparation will look quite tedious at first sight, but you will understand the beauty of it by the end of the tutorial ;)

Prepare your remote repository

First, you should **create an account on GitHub** (it is free) if you do not have one. Your account basically allows you to have a space where to store your own *repositories*.

A repository is basically just a folder that contains subfolders and files. But this folder is *versioned*, thanks to Git. This means that **you can precisely follow every change that has been made within this folder since its creation**. In other word you have access to every *version* of the folder since its creation, which means every version of the software in the case of a computer program. And if at some point you make a modification of the code that break everything, you can safely go back to the previous version.

What about our precise case?

We noticed before that there is already a *Physik Instrument* plugin repository, it is then not necessary to create another one. We would rather like to *modify* it, and add a new file that would deal with our E870 controller. Let first make a copy of this repository into our account. In the technical jargon of Git, we say that we will make a *fork* of the repository. The term *fork* images the fact that we will make a new history of the evolution of the folder. By forking the repository into our account, we will keep track of *our modifications* of the folder, and the original one can follow another trajectory.

To fork a repository, follow this procedure:

- Log in to your GitHub account

Fig. 7.140: E870 manual: how to configure the demultiplexer.

Fig. 7.141: Script to change the controlled axis.

- Go to the original repository (called the *upstream repository*) (in our case the repository is stored by the PyMoDAQ organisation) and click *Fork*.

Fig. 7.142: How to fork a repository through GitHub.

GitHub will create a copy of the repository on our account (*quantumm* here).

Fig. 7.143: Our *PI remote* repository (in our GitHub account). The red boxes indicate how to find the GitHub address of this repository.

This repository stored on our account is called the *remote repository*.

Prepare your local repository

First you should [install Git](#) on your machine.

Then we will make a local copy of our remote repository, that we will call the *local repository*. This operation is called *cloning*. Click the *Code* icon and then copy in the clipboard the HTTPS address.

In your home folder, create a folder called *local_repository* and cd into it by executing in your terminal

```
cd C:\Users\<OS username>\local_repository\
```

(actually you can do the following in the folder you like).

Then clone the repository with the following command

```
git clone https://github.com/<GitHub username>/pymodaq_plugins_physik_instrumente.git
```

this will create a folder at your current location. Go into it

```
cd pymodaq_plugins_physik_instrumente
```

Notice that we just downloaded the content of the remote repository.

We will also create a new *branch* named *E-870* with the following command

```
git checkout -b E-870
```

Now if you execute the command

```
git status
```

the output should start with “On branch E-870”.

Install your package in edition mode

We now enter the Python world and talk about a *package* rather than a repository, but we are actually still talking about the same folder!

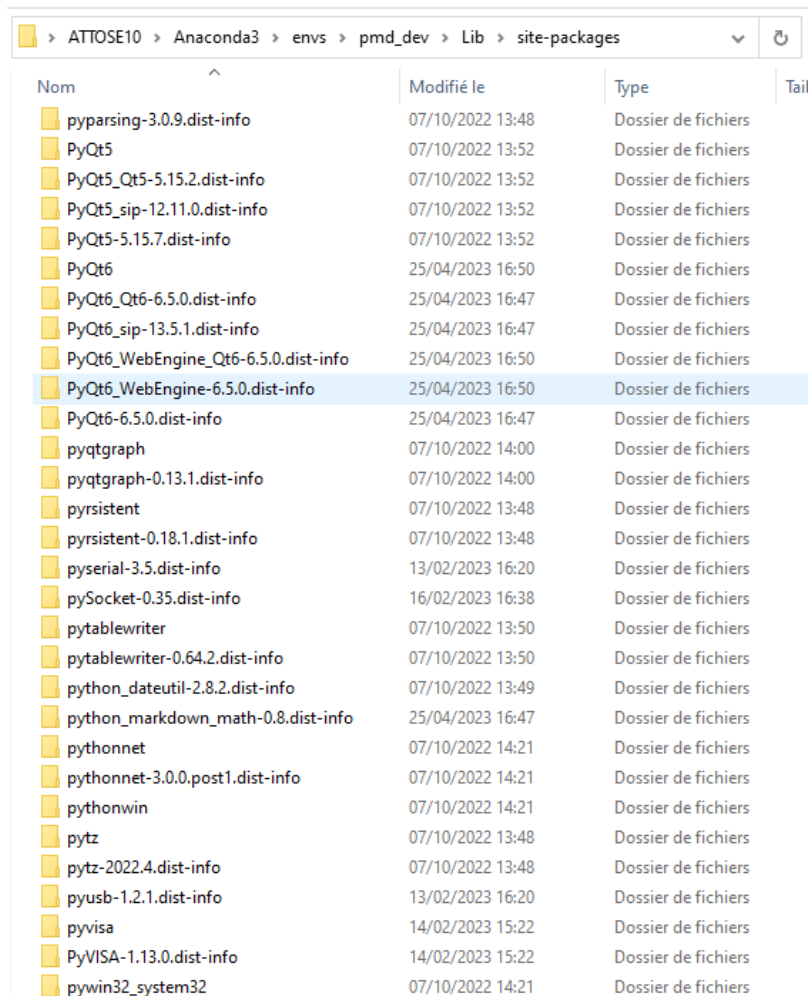
Still in your terminal, check that your Python environment *pmd_dev* is activated, and stay at the root of the package. Execute the command

```
(pmd_dev) C:\Users\<OS username>\local_repository\pymodaq_plugins_physik_instrumente>pip
install -e .
```

Fig. 7.144: Illustration of the operations between the different repositories.

Understanding this command is not straightforward. In your Python environment, there exists an important folder called *site-packages* that you should find at the following path

C:\Users\<OS_username>\Anaconda3\envs\dev_pid\Lib\site-packages



Nom	Modifié le	Type	Tail
pyarsing-3.0.9.dist-info	07/10/2022 13:48	Dossier de fichiers	
PyQt5	07/10/2022 13:52	Dossier de fichiers	
PyQt5_Qt5-5.15.2.dist-info	07/10/2022 13:52	Dossier de fichiers	
PyQt5_sip-12.11.0.dist-info	07/10/2022 13:52	Dossier de fichiers	
PyQt5-5.15.7.dist-info	07/10/2022 13:52	Dossier de fichiers	
PyQt6	25/04/2023 16:50	Dossier de fichiers	
PyQt6_Qt6-6.5.0.dist-info	25/04/2023 16:47	Dossier de fichiers	
PyQt6_sip-13.5.1.dist-info	25/04/2023 16:47	Dossier de fichiers	
PyQt6_WebEngine_Qt6-6.5.0.dist-info	25/04/2023 16:50	Dossier de fichiers	
PyQt6_WebEngine-6.5.0.dist-info	25/04/2023 16:50	Dossier de fichiers	
PyQt6-6.5.0.dist-info	25/04/2023 16:47	Dossier de fichiers	
pyqtgraph	07/10/2022 14:00	Dossier de fichiers	
pyqtgraph-0.13.1.dist-info	07/10/2022 14:00	Dossier de fichiers	
pyrsistent	07/10/2022 13:48	Dossier de fichiers	
pyrsistent-0.18.1.dist-info	07/10/2022 13:48	Dossier de fichiers	
pyserial-3.5.dist-info	13/02/2023 16:20	Dossier de fichiers	
pySocket-0.35.dist-info	16/02/2023 16:38	Dossier de fichiers	
pytablewriter	07/10/2022 13:50	Dossier de fichiers	
pytablewriter-0.64.2.dist-info	07/10/2022 13:50	Dossier de fichiers	
python_dateutil-2.8.2.dist-info	07/10/2022 13:49	Dossier de fichiers	
python_markdown_math-0.8.dist-info	25/04/2023 16:47	Dossier de fichiers	
pythonnet	07/10/2022 14:21	Dossier de fichiers	
pythonnet-3.0.0.post1.dist-info	07/10/2022 14:21	Dossier de fichiers	
pythonwin	07/10/2022 14:21	Dossier de fichiers	
pytz	07/10/2022 13:48	Dossier de fichiers	
pytz-2022.4.dist-info	07/10/2022 13:48	Dossier de fichiers	
pyusb-1.2.1.dist-info	13/02/2023 16:20	Dossier de fichiers	
pyvisa	14/02/2023 15:22	Dossier de fichiers	
PyVISA-1.13.0.dist-info	14/02/2023 15:22	Dossier de fichiers	
pywin32_system32	07/10/2022 14:21	Dossier de fichiers	

Fig. 7.145: Content of the *site-packages* folder of our *pmd_dev* environment.

The subfolders that you find inside correspond to the Python packages that are installed within this environment. A general rule is that **you should never modify manually anything in this folder**. Those folders contain the exact versions of each package that is installed in our environment. If we modify them in a dirty way (not versioned), we will very fast loose the control about our modifications. The *edition* option “-e” of *pip* is the solution to work in a clean way, it allows to simulate that our package is installed in the environment. This way, during the development period of our plugin, we can safely do any modification in our folder C:\Users\<OS_username>\Vocal_repository\pymodaq_plugins_physik_instrumente (referred to by the “.” in the command) and it will behave as if it was in the *site-packages*. To check that this last command executed properly, you can check that you have a file called *pymodaq_plugins_physik_instrumente.egg-link* that has been created in the *site-packages* folder. Note that *pip* knows with which Python environment to deal with because we have activated *pmd_dev*.

Open the package with an adapted IDE

In this section we will work not only with a simple script, but within a Python project that contains multiple files and that is much more complex than a simple script. For that Spyder is not so well adapted. In this section we will present **PyCharm** because it is free and very powerful, but you can probably find an equivalent one.

Once it is opened, go to *File > New project*. Select the repository folder and the Python interpreter.

Fig. 7.146: Start a project with PyCharm. You have to select the main folder that you will work with, and the Python interpreter corresponding to your environment.

You can for example configure the interface so that it looks like the following figure.

Fig. 7.147: PyCharm interface. **Left panel:** tree structure of the folders that are included in the PyCharm project. **Center:** edition of the file. **Right panel:** structure of the file. Here you find the different methods and argument of the Python class that are defined in the file. **Bottom:** different functionalities that are extremely useful: a Python console, a terminal, a debugger, integration of Git...

In the left panel, you will find the folder corresponding to our repository, so that you can easily open the files you are interested in. We will also add in the project the PyMoDAQ core folder, so that we can easily call some entry points of PyMoDAQ. To do so, go to *File > Open* and select the PyMoDAQ folder. Be careful to not get lost in the tree structure, you have to go select the folder that is in the good environment. In this case `C:\Users\<OS username>\Anaconda3\envs\pmd_dev\Lib\site-packages\pymodaq` (in particular, do not mistake with the *site-packages* of the base Anaconda environment that is located at `C:\Users\<OS username>\Anaconda3\Lib\site-packages`), click *OK* and then *Attach*.

The *pymodaq* folder should now appear in the left panel, navigate within it, open and *Run* (see figure) the file *pymodaq > daq_move > daq_move_main.py*. This is equivalent to execute the *daq_move* command in a terminal. Thus you should now see the GUI of the DAQ_Move.

Debug of the original plugin

As we have noticed before, a lot of things were already working in the original plugin. It is now time to analyse what is happening. For that, we will use the *debugger* of our IDE, which is **an indispensable to debug PyMoDAQ**. You will save a lot of time by mastering this tool! And it is very easy to use.

Let us now open the *daq_move_PI.py* file. This file defines a class corresponding to the original *PI* plugin, and you can have a quick look at the methods inside using the *Structure* panel of PyCharm. Basically, most of the methods of the class are triggered by a button from the user interface, as is illustrated in the following figure.

Fig. 7.148: Each action of the user on the UI triggers a method of the instrument class.

During our first test of the plugin, earlier in this tutorial, we noticed that things went wrong at the moment we click the *Initialize* button, which corresponds to the *ini_stage* method of the *DAQ_Move_PI* class. We will place inside this method some *breakpoints* to analyse what is going on. To do so, you just have to click within the *breakpoints column* at the line you are interested in. A red disk will appear, as illustrated by the next capture.

When you run a file in *DEBUG* mode (bug button instead of play button), it means that PyCharm will execute the file until it finds an activated breakpoint. It will then stop the execution and wait for orders: you can then resume the program up to the next breakpoint, or execute line by line, rerun the program from the beginning...

When you run the *DEBUG* mode, notice that a new *Debug* panel appears at the bottom of the interface. The *View breakpoints* button will popup a window so that you see where are the breakpoints *within all your PyCharm project*,

Fig. 7.149: See the breakpoints inside your PyCharm project.

that is to say within all the folders that you *attached* to your project, and that are present in the tree structure of the *Project panel*. You can also deactivate a breakpoint, in that case it will be notified with a red circle.

Fig. 7.150: Execute PyMoDAQ in DEBUG mode.

Let us now run in DEBUG mode the *daq_move_main.py* file. We select the *PI* plugin (not the *PI E870*), the good controller, and initialize. PyCharm stops the execution at the first breakpoint and highlight the corresponding line in the file. This way we progress step by step up to “sandwiching” the line that triggers the error with breakpoints. Looking at the value of the corresponding variable, we found again the *Unknown command (2)* error message that we already had in the PyMoDAQ log file.

Fig. 7.151: Find the buggy line. The breakpoint line 163 is never reached. The value of the *self.controller.gcscommands.axes* variable is *Unknown command (2)*.

Let’s go there to see what happens. We can attach the *pipython* package to our PyCharm project and look at this *axes* attribute. In this method we notice the call to the *qSAI* method, which is NOT supported by our controller! We now have a precise diagnosis of our bug :)

Write the class for our new instrument

Coding a PyMoDAQ plugin actually consists in writing a Python class with specific conventions such that the PyMoDAQ core knows where to find the installed plugins and where to call the correct methods.

The [PyMoDAQ plugins template](#) repository is here to help you follow those conventions and such that you have to do the minimum amount of work. Let see what it looks like!

The *src* directory of the repository is subdivided into three subfolders

- *daq_move_plugins* which stores all the instruments corresponding to actuators.
- *daq_viewer_plugins*, which stores all the instruments corresponding to detectors. It is itself divided into subfolders corresponding to the dimensionality of the detector.
- *hardware*, within which you will find Python wrappers (optional).

Within each of the first two subfolders, you will find a Python file defining a class. In our context we are interested in the one that is defined in the first subfolder.

As you can see the structure of the instrument class is already coded. What we have to do is to follow the comments associated to each line, and insert the scripts we have developped in a previous section (see *gold route*) in the right method.

There are *naming conventions* that should be followed:

- We already mentioned that the name of the package should be *pymodag-plugins-<company name>*. Do not forget the “s” at “plugins” ;)
- The name of the file should be *daq_move_XXX.py* and replace *XXX* by whatever you like (something that makes sense is recommended ;)
- The name of the class here should be *DAQ_Move_XXX*.
- The name of the methods that are already present in the template should be kept as it is.

Fig. 7.152: The *axes* attribute calls the *SAI?* GCS command that is not supported by the E870 controller.

main ▾ pymodag_plugins_template / src / pymodag_plugins_template / Go to file

seb5g	Update daq_move_Template.py	c4a8e3f on Feb 6	History
..			
daq_move_plugins	Update daq_move_Template.py	3 months ago	
daq_viewer_plugins	simplifies hidden dependencies on daq_utils	3 months ago	
hardware	implemented the src layout	3 years ago	
VERSION	Update VERSION	last year	
init.py	simplifies hidden dependencies on daq_utils	3 months ago	

Fig. 7.153: Tree structure of the plugin template repository.

```

1 from pymodag.control_modules.move_utility_classes import DAQ_Move_base, comon_parameters_fun, main # common set of parameters for all actuators
2 from pymodag.daq_utils.daq_utils import ThreadCommand # object used to send info back to the main thread
3 from pymodag.daq_utils.parameter import Parameter
4
5 class PythonWrapperOfYourInstrument:
6     # TODO Replace this fake class with the import of the real python wrapper of your instrument
7     pass
8
9
10 class DAQ_Move_Template(DAQ_Move_base):
11     """Plugin for the Template Instrument
12
13     This object inherits all functionality to communicate with PyMoDAQ Module through inheritance via DAQ_Move_base
14     It then implements the particular communication with the instrument
15
16     Attributes:
17     -----
18     controller: object
19         The particular object that allow the communication with the hardware, in general a python wrapper around the
20         hardware library
21     # TODO add your particular attributes here if any
22
23     """
24     _controller_units = 'whatever' # TODO for your plugin: put the correct unit here
25     is_multiaxes = False # TODO for your plugin set to True if this plugin is controlled for a multiaxis controller
26     axes_names = ['Axis1', 'Axis2'] # TODO for your plugin: complete the list
27     _epsilon = 0.1 # TODO replace this by a value that is correct depending on your controller
28
29     params = [ # TODO for your custom plugin: elements to be added here as dicts in order to control your custom stage
30               ] + comon_parameters_fun(is_multiaxes, axes_names, epsilon=_epsilon)
31     # _epsilon is the initial default value for the epsilon parameter allowing pymodag to know if the controller reached
32     # the target value. It is the developer responsibility to put here a meaningful value
33
34     def ini_attributes(self):
35         # TODO declare the type of the wrapper (and assign it to self.controller) you're going to use for easy
36         # autocompletion
37         self.controller: PythonWrapperOfYourInstrument = None
38
39         # TODO declare here attributes you want/need to init with a default value
40         pass
41
42     def get_actuator_value(self):
43         """Get the current value from the hardware with scaling conversion.
44
45         Returns
46         -----

```

Fig. 7.154: Definition of the DAQ_Move_Template class.

Note: Be careful that in the package names, the separator is “-”, whereas in file names, the separator is “_”.

The name of the methods is quite explicit. Moreover, the *docstrings* are here to help you understand what is expected in each method.

Note: In Python, a method’s name should be lowercase.

Go to the *daq_move_plugins* folder, you should find some files like *daq_move_PI.py*, which correspond to the other plugins that are already present in this package.

With a right click, we will create a new file in this folder that we will call *daq_move_PI_E870.py*. Copy the content of the *daq_move_Template.py* file and paste it in the newly created file.

Change the name of the class to *DAQ_Move_PI_E870*.

Run again the *daq_move_main.py* file.

You should now notice that our new instrument is already available in the list! This is thanks to the naming conventions. However, the initialization will obviously fail, because for now we did not input any logic in our class.

Before we go further, let us configure a bit more PyCharm. We will first fix the maximum number of characters per lign. Each Python project fixes its own convention, so that the code is easier to read. For PyMoDAQ, the convention is **120 characters**. Go to *File > Settings > Editor > Code Style* and configure *Hard wrap* to 120 characters.

Introduction of the class

We call the *introduction of the class* the code that is sandwiched between the *class* keyword and the first method definition. This code will be executed after the user selected the instrument he wants to use through the *DAQ_Move* UI.

This part of the code from the original plugin was working, so let’s just copy-paste it, and adapt a bit to our case.

Fig. 7.155: Introduction of the class of our PI E870 instrument.

First, it is important that we comment the context of this file, this can be done in the *docstring* attach to the class, PyMoDAQ follows the [Numpy style](#) for its documentation

Notice that the import of the wrapper is very similar to what we have done in the gold route. However, we do not call anymore the *InterfaceSetupDlg()* method that was popping up a window. We rather use the *EnumerateUSB()* method to get the list of the addresses of the plugged controllers, which will then be sent in the parameter panel (in the item named *Devices*) of the *DAQ_Move* UI. We now understand precisely the sequence of events that makes the list of controller addresses available just after we have selected our instrument.

Notice that in the class declaration not all the parameters are visible. Most of them are declared in the *comon_parameters_fun* that declares all the parameters that are common to every plugin. But if at some point you need to add some specific parameter for your instrument, you just have to add an element in this *params* list, and it will directly be displayed and controllable through the *DAQ_Move* UI! You should fill in a *title*, a *name*, a *type* of data, a *value* ... You will find this kind of tree everywhere in the PyMoDAQ code. Copy-paste the first lign for exemple and see what happens when you execute the code ;)

To modify the value of such a parameter, you will use something like

```
self.settings.child('multiaxes', 'axis').setValue(2)
```

Here we say “in the parameter tree, choose the *axis* parameter, in the *multiaxes* group, and attribute him the value 2 “

Note: `self.settings` is a *Parameter* object of the `pyqtgraph` library.

Get the value of this parameter will be done with

```
self.settings['multiaxes', 'axis']
```

ini_stage method

As mentioned before, the *ini_stage* method is triggered when the user click the *Initialization* button. It is here that the communication with the controller is established. If everything works fine, the LED will turn green.

Fig. 7.156: *ini_stage* method of our PI E870 instrument class.

Compared to the initial plugin, we simplified this method by removing the functions that were intended for close-loop operation. Plus we only consider the USB connexion. The result is that our controller initializes correctly now: the LED is green!

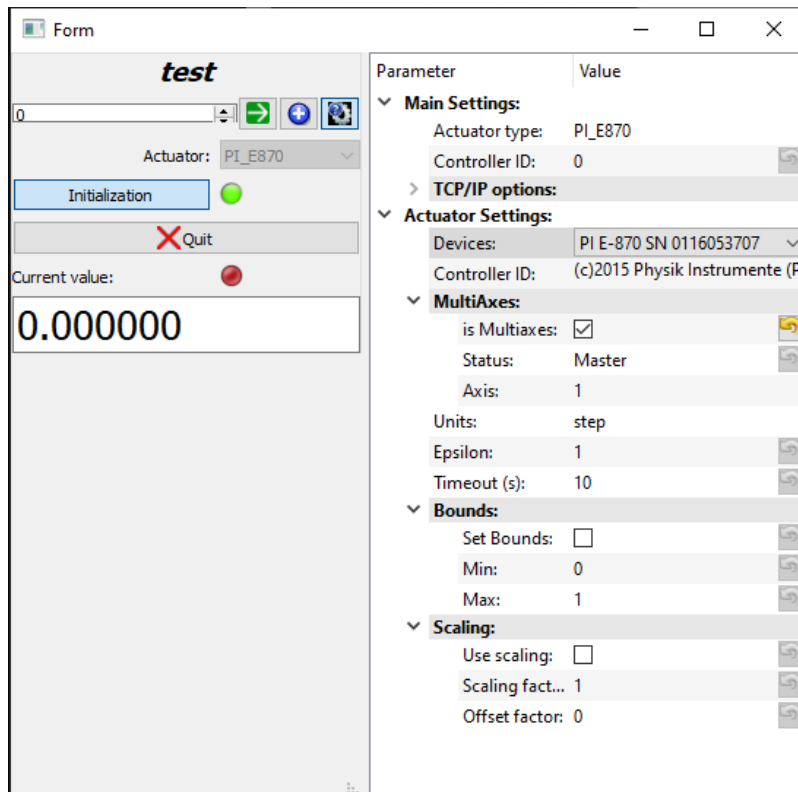


Fig. 7.157: Now our controller initializes correctly.

commit_settings method

Another important method is *commit_settings*. This one contains the logic that will be triggered when the user modifies some value in the parameter tree. Here will be implemented the change of axis of the controller, by changing the configuration of the demultiplexer with the *MOD GCS* command (see the gold route).

Fig. 7.158: *commit_settings* method of our PI E870 instrument class. Implementation of a change of axis.

move_rel method

Finally, the *move_rel* method, that implements a relative move of the actuator is quite simple, we just use the *OSM* command that we found when we studied the DLL with a simple script.

Fig. 7.159: *move_rel* method of our PI E870 instrument class. Implementation of a relative move.

We can now test the *Rel + / Rel -* buttons, a change of axis... it works!

There is still minor methods to implement, but now you master the basics of the instrument plugin development ;)

Commit our changes with Git

Now that we have tested our changes, we can be happy with this version of our code. We will now **stamp this exact content of the files**, so that in the future, we can at any time fall back to this working version. You should see Git as your guarantee that you will never lost anything of your work.

At the location of our local repository, we will now use this Git command

```
C:\Users\<OS username>\local_repository\pymodaq_plugins_physik_instrumente>git diff
```

you should get something that looks like this

This Git command allows us to check precisely the modifications we have done, which is called a *diff*.

In the language of Git, we stamp a precise state of the repository by doing a *commit*

```
C:\Users\<OS username>\local_repository\pymodaq_plugins_physik_instrumente>git commit -am  
"First working version of the E870 controller plugin."
```

Within the brackets, we leave a comment to describe the changes we have made.

Then, with the *git log* command, you can see the history of the evolution of the repository

```
C:\Users\<OS username>\local_repository\pymodaq_plugins_physik_instrumente>git log
```

Push our changes to our remote repository

We have now something that is working locally. That is great, but what if at some point, the computer of my experiment suddenly crashes? What if I want to share my solution to a colleague that have the same equipment?

Would not it be nice if I could command my controller on any machine in the world with a few command lines?
:O

It is for those kind of reasons that it is so efficient to work with a remote server. It is now time to benefit from our careful preparation! Sending the modifications on our remote repository is done with a simple command

```
C:\Users\<OS username>\local_repository\pymodaq_plugins_physik_instrumente>git push
```

In the Git vocabulary, *pushing* means that you send your work to your *remote repository*. If we go on our remote server on GitHub, we can notice that our repository has actually been updated!

From now on, anyone who has an internet connexion have access to this precise version of our code.

Note: You may wonder how Git knows where to push? This has been configured when we cloned our remote repository. You can ask what is the current address configured of your remote repository (named *origin*) with the *git remote -v* command.

```

-class DAQ_Move_PI(DAQ_Move_base):
-    """
-    Plugin using the pipython package wrapper. It is compatible with :
-    DLLDEVICES = {
-    'PI_GCS2_DLL': ['C-413', 'C-663.11', 'C-863.11', 'C-867', 'C-877', 'C-884', 'C-885', 'C-887',
-                  'C-891', 'E-517', 'E-518', 'E-545', 'E-709', 'E-712', 'E-723', 'E-725',
-                  'E-727', 'E-753', 'E-754', 'E-755', 'E-852B0076', 'E-861', 'E-870', 'E-871',
-                  'E-873', 'C-663.12'],
-    'C7XX_GCS_DLL': ['C-702', ],
-    'C843_GCS_DLL': ['C-843', ],
-    'C848_DLL': ['C-848', ],
-    'C880_DLL': ['C-880', ],
-    'E816_DLL': ['E-621', 'E-625', 'E-665', 'E-816', 'E816', ],
-    'E516_DLL': ['E-516', ],
-    'PI_Mercury_GCS_DLL': ['C-663.10', 'C-863.10', 'MERCURY', 'MERCURY_GCS1', ],
-    'PI_HydraPollux_GCS2_DLL': ['HYDRA', 'POLLUX', 'POLLUX2', 'POLLUXNT', ],
-    'E7XX_GCS_DLL': ['DIGITAL PIEZO CONTROLLER', 'E-710', 'E-761', ],
-    'HEX_GCS_DLL': ['HEXAPOD', 'HEXAPOD_GCS1', ],
-    'PI_G_GCS2_DLL': ['UNKNOWN', ],
-    """
+class DAQ_Move_PI_E870(DAQ_Move_base):
+    """Minimalistic plugin for the PI E870 4G controller with PiezoMike actuators.
-
-    _controller_units = 'mm' # dependent on the stage type so to be updated accordingly using self.controller_units =
new_unit
+    Use the pipython package wrapper.
+    It works in open loop. There is no referencing. It considers only relative moves.
+    It does not consider the daisy chain option: only one controller.
+    Only USB connexion is implemented.
+    Tested with PI_E870_4G: we consider 4 axes.
+    """
+    _controller_units = 'step'
+    gcs_device = GCSDevice()
+    devices = gcs_device.EnumerateUSB()
+    devices.extend(gcs_device.EnumerateTCPIPDevices())
+
+    devices.extend([str(port) for port in list(list_ports.comports())])
+    devices = gcs_device.EnumerateUSB() # we only look for the controllers that are plugged with USB.
+    is_multiaxes = True
+    stage_names = []
+    _epsilon = 0.01
+    axes_names = [1, 2, 3, 4]
+    _epsilon = 1
+
+    params = [
+        {'title': 'Connection type:', 'name': 'connect_type', 'type': 'list',
+         'value': 'USB', 'values': ['USB', 'TCP/IP', 'RS232']},
+        {'title': 'Devices:', 'name': 'devices', 'type': 'list', 'values': devices},
+        {'title': 'Daisy Chain Options:', 'name': 'dc_options', 'type': 'group', 'children': [
+            {'title': 'Use Daisy Chain:', 'name': 'is_daisy', 'type': 'bool', 'value': False},
+            {'title': 'Is master?:', 'name': 'is_daisy_master', 'type': 'bool', 'value': False},
+            {'title': 'Daisy Master Id:', 'name': 'daisy_id', 'type': 'int'},
+            {'title': 'Daisy Devices:', 'name': 'daisy_devices', 'type': 'list'},
+            {'title': 'Index in chain:', 'name': 'index_in_chain', 'type': 'int', 'enabled': True}],
+        {'title': 'Use Joystick:', 'name': 'use_joystick', 'type': 'bool', 'value': False},
+        {'title': 'Closed loop?:', 'name': 'closed_loop', 'type': 'bool', 'value': True},
+        {'title': 'Controller ID:', 'name': 'controller_id', 'type': 'str', 'value': '', 'readonly': True},
+        {'title': 'Axis Info:', 'name': 'axis_infos', 'type': 'group', 'children': [
+            {'title': 'Min:', 'name': 'min', 'type': 'float'},
+            {'title': 'Max:', 'name': 'max', 'type': 'float'},
+        ]},
+    ] + common_parameters_fun(is_multiaxes, stage_names, epsilon=_epsilon)
+    ] + common_parameters_fun(is_multiaxes, axes_names, epsilon=_epsilon)

```

Fig. 7.160: Answer to the *git diff* command in a terminal. Here are the modifications of the `daq_move_PI_E870.py` file. In red are the lines that have been deleted, in green the lines that have been added.

Fig. 7.161: Answer to the *git log* command in a terminal.

Fig. 7.162: The *git push* command updated our remote repository.

Pull request to the upstream repository

But this is not the end! Since we are very proud of our new plugin, why not make all the users of PyMoDAQ benefit from it? Why not propose our modification to the official *pymodaq_plugin_physik_instrumente* repository?

Again, since we prepared properly, it is now a child play to do that. In the Git vocabulary, we say that we will do a *pull request*, often abbreviated as PR. This can be done through the interface of GitHub. Log in to your account, go to the repository page and click, in the *Pull request* tab, the *Create pull request* button.

You have to be careful to select properly the good repositories and the good branches. Remember that in our case we created a *E-870* branch.

Fig. 7.163: The GitHub interface to create a PR.

Leave a message to describe your changes and submit. Our pull request is now visible [on the upstream repository](#).

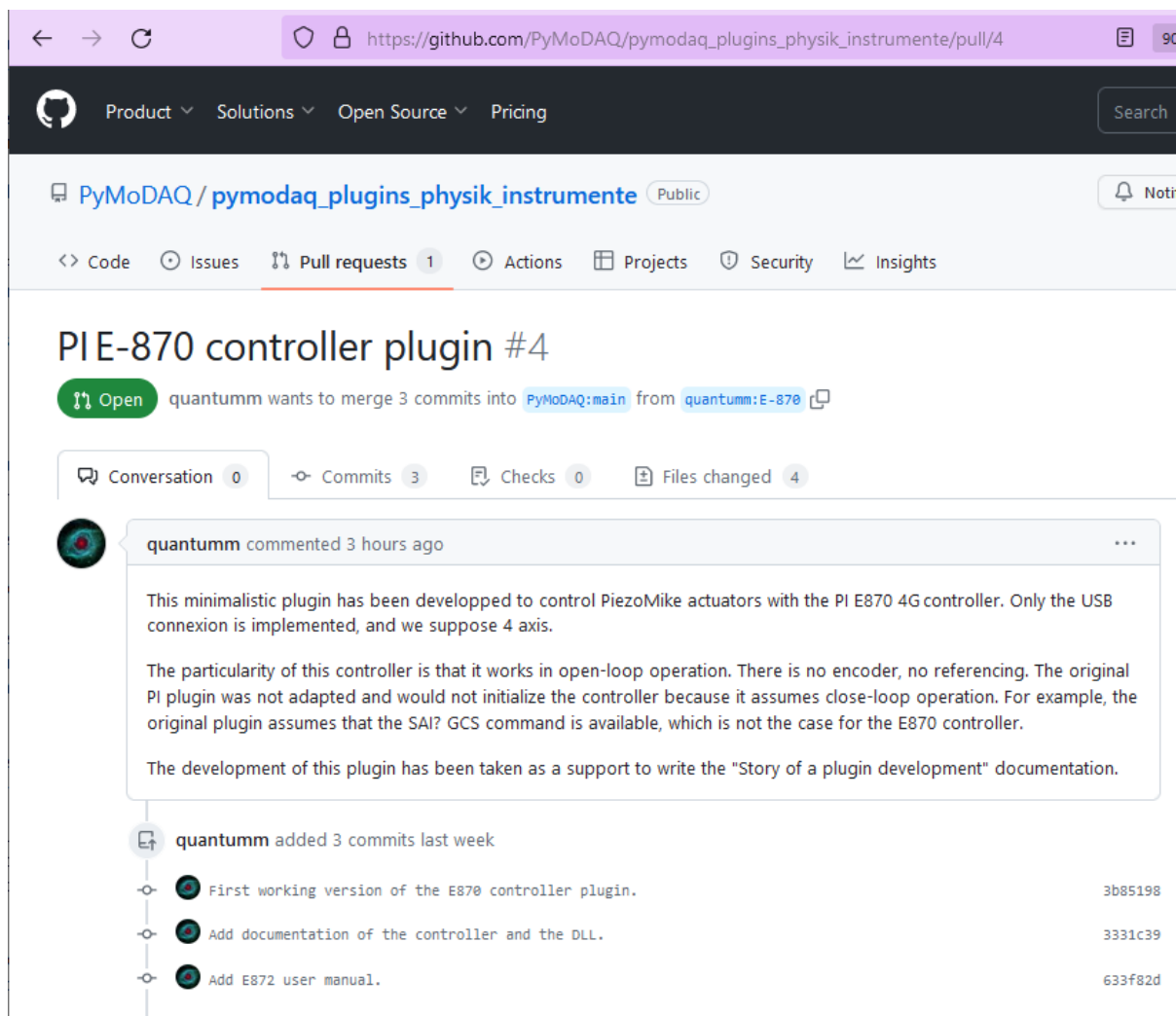


Fig. 7.164: Our pull request in the upstream repository.

This opens a space where you can discuss your changes with the owner of the repository. It will be his decision to accept or not the changes that we propose. Let us hope that we will convince him! :) Often these discussions will lead to a significant improvement of the code.

Conclusion

That's it!

We have tried, with this concrete example, to present the global workflow of an instrument plugin development, and the most common problems you will face. Do not forget that you are not alone: ask for help, it is an other way to meet your colleagues!

We have also introduce a software toolbox for Python development in general, that we sum up in the following table. They are all free of charge. Of course this is just a suggestion, you may prefer different solutions. We wanted to present here the main types of software you need to develop efficiently.

Software function	Solution presented
Python environment manager	Anaconda
Python package manager	pip
Python IDE	Spyder / PyCharm
Version control software	Git
Repository host	GitHub

Finally, remember that while purchasing an instrument, it is important to check what your supplier provides as a software solution (Python wrapper, Linux drivers...). This can save you a lot of time!

7.5.5 How to contribute to PyMoDAQ's documentation?

In this tutorial we will learn how to contribute to PyMoDAQ's documentation. This is quite an advanced topic so we consider that you know quite well Python and PyMoDAQ ecosystem.

The documentation of PyMoDAQ

There are several levels of documentation of the project that we introduce in the following sections.

Documentation of the source code: docstrings

The documentation of the source code is done using comments in the source files that are called *docstrings*. This documentation is addressed to the developers and is very precise. Typically, it will explain the job of a particular method, give the type of its arguments, what it returns...

Fig. 7.165: Docstring of the *move_abs* method in the *daq_move_Template.py* file.

This kind of documentation is standardized. PyMoDAQ follows the [Numpy docstrings style](#). Following those conventions permits to generate automatically the *Library Reference*.

Tests

At each modification of the source code of PyMoDAQ, a series of tests is launched automatically. This is done to ensure that the modification proposed does not have an unexpected effect and does not break the rest of the code. This development practice is indispensable to ensure its stability. A big effort has been devoted to testing in the version 4 of PyMoDAQ.

The files defining the tests are located in the `/tests` directory at the root of the repository.

Most of those tests simulate a user interacting with PyMoDAQ UI, pressing buttons and so on, and verify that everything is working as expected.

Reading those tests (which is not straightforward ;)) allows to get a global picture of what the application is doing.

Website

Finally, there is the website that you are reading right now. This documentation is of higher level than the previous ones, easier to read for a human! It is then adapted mostly to an introduction of PyMoDAQ to users.

This tutorial intends to present the workflow to contribute to the improvement of this website.

Sphinx

You may have noticed that most of Python librairies, share a common presentation of their website, this is because they all use `Sphinx` as a documentation generator.

Sphinx uses `reStructuredText`, which is the standard lightweight language to write documentation within the Python community.

Using Sphinx saves a lot of time because you just have to care about the *content* of your documentation, Sphinx will then render it as a beautiful PDF file... or even a website, like the one you are reading right now!

The folder within which there is a `conf.py` file is the *source directory* of a Sphinx project. In our case this directory is `PyMoDAQ/docs/src`.

Notice that this directory is included in PyMoDAQ repository. Therefore, contributing to the documentation, from the point of view of Git, is exactly the same thing as contributing to the source code: we will modify files in the repository.

Note: The `/docs` directory of PyMoDAQ is located at the root of the repository, aside with the `/src` directory. When you install the `pymodaq` package, what will be copied in the *site-packages* of your Python environment in the `PyMoDAQ/src/pymodaq` folder. Therefore, all the folders that are upstream from this one (including `/docs`) will not be copied in the *site-packages*. This is what we want, it would be useless to have all this documentation, intended for humans, in the *site-packages*.

Preparation

Let's prepare properly our workspace. We consider that you have a GitHub account, that you know the basics about its usage, and that you have already a remote repository (you have forked PyMoDAQ in your GitHub account).

First we need to know on which branch of the `upstream repository` we will work. If we want to contribute to the core of PyMoDAQ, we should send a pull request to the `pymodaq-dev` branch.

Note:

The important branches of the PyMoDAQ repository are as follow:

- **main** is the last stable version. This branch is maintained by the owner of the repository, and we should not send a pull request directly to it.
 - **pymodaq-dev** is the development branch, which is ahead of the *main* branch (it contains more commits than the *main* branch. External contributions should be send on this branch. The owner of the repository will test all the changes that has been suggested in the *pymodaq-dev* branch before sending them into the *main* branch.
 - **pymodaq_v3** concerns the version 3.
-

Let's *create and activate a new Python environment*, that we will call *pmd_dev* in this tutorial.

Let's now clone this specific branch on our local machine. We will call our local repository *pmd4_write_documentation_tutorial*.

```
git clone --branch pymodaq-dev https://github.com/PyMoDAQ/PyMoDAQ.git
pmd4_write_doc_tutorial
```

and cd into it

```
cd pmd4_write_doc_tutorial
```

We have to change the configuration of *origin* so that our local repository is linked to our remote repository, and not to the upstream repository.

```
git remote set-url origin https://github.com/<your GitHub name>/PyMoDAQ.git
```

Note: *origin* is an alias in Git that should target your remote repository. It specifies where to push your commits.

We can check that it has been taken into account with

```
git remote -v
```

We will now create a new branch from *pymodaq-dev* so that we can isolate our changes. We call it *write-doc-tutorial*.

```
git checkout -b write-doc-tutorial
```

Finally, install our local repository in edition mode in our Python environment

```
(pmd_dev) >pip install -e .
```

We can now safely modify our local repository.

Build the website locally

Since the source of the website (in */docs/src*) is included in the PyMoDAQ repository, it means that we have everything needed to build it locally!

Some additional packages are necessary to install, in particular *sphinx*, *docutils*, *numpydoc*... Those guys are listed in the *requirements.txt* file in the */docs* directory. Let's go into it and execute the command

```
(pmd_dev) >pip install -r requirements.txt
```

Still in the */docs* folder (where you should have a *make.bat* file) execute

```
make html (. \make html on windows powershell)
```

This will run *Sphinx* that will build the website and put it into the newly created *docs/_build* folder. Open the */docs/_build/html/index.html* file with your favorite navigator. You just build the website locally!

Fig. 7.166: Local build of the PyMoDAQ website.

Add a new tutorial

Let's take a practical case, and suppose we want to add a tutorial about "How to contribute to PyMoDAQ's documentation?" ;)

Fig. 7.167: Sphinx source directory. It contains *index.rst* which defines the welcome page of the website and the table of contents. It contains also the *conf.py* file which defines the configuration of Sphinx. In the subfolders are others *.rst* file defining other pages. The */image* folder is where one can store the images that are included in the pages.

The *index.rst* file defines the welcome page of the website, add also the table of contents that you see on the left column.

Fig. 7.168: In the *index.rst* file, the *toctree* tag defines the first level of the table of contents.

We clearly have to go in the *tutorial* folder. Here we found the *plugin_development.rst* file where is written the tutorial "Story of an instrument plugin development".

Let's just create a new *.rst* file named *write_documentation.rst*. We will copy the introduction of the other file, just replacing the name of the label (first line) and the title.

```
.. _write_documentation:

How to contribute to PyMoDAQ's documentation?
=====
```

In the *tutorials.rst* file, there is another *toctree* tag which defines the second level of the table of contents within the *Tutorials* section. We have to say that there is a new entry. Notice that it is here that the label at the first line of the file is important.

```
Tutorials
=====

.. toctree::
    :maxdepth: 5
    :caption: Contents:

    tutorials/plugin_development
    tutorials/write_documentation
```

Save this file and compile again with Sphinx in the */docs* directory

`make html` (`.\make html` on windows powershell)

and refresh the page in the navigator. Our new tutorial is already included in the website, and the table of contents has been updated!

We just have to fill the rest of the page with what we have to say! We will introduce a bit the RST language in the following section.

Fig. 7.169: First compilation of our new tutorial.

reStructuredText (RST) language

Here we give a brief overview of the RST language. Here is the [full documentation about RST](#).

Page structure

```
Title
=====

Section
-----

Lorem ipsum lorem ipsum.

Subsection
+++++++

Lorem ipsum lorem ipsum. Lorem ipsum lorem ipsum.
```

List

```
* First item

    * First item of nested list
    * Second item of nested list

* Second item
```

External link (URL)

```
`PyMoDAQ repository`__
__ https://github.com/PyMoDAQ/PyMoDAQ
```

Integrate an image

```
.. _fig_label
.. figure:: /image/write_documentation/my_image.svg
    :width: 600

Caption of the figure.
```

The images are saved in the `/src/image` folder and subfolders.

Notice that you can directly integrate SVG images.

Note: Be careful that the extensions of your files **should be lowercase**. The Windows operating system does not differentiate file extensions .PNG and .png for example (it is not case sensitive). If you build the documentation locally on Windows, it could render it without problem, while when compiled with a Linux system (what will be done on the server) your paths can be broken and your images not found.

Cross-referencing

If we want to refer to the image from the previous section:

```
:numref: `fig_label`
```

Note: Note that the underscore disappeared.

If we want to refer to another page of the documentation:

```
:ref: `text to display <label at the begining of the page>`
```

for example to refer to the installation page, we will use

```
:ref: `install PyMoDAQ <section_installation>`
```

Glossary terms

You may have notice the *Glossary Terms* page in the page of contents. This is a kind of dictionary dedicated to PyMoDAQ documentation. There are defined terms that are used frequently in the documentation. Referring to those term is then very simple

```
:term: `the glossary term`
```

Browse the already written RSTfiles to get some examples ;)

Submit our documentation to the upstream repository

We are now happy with the content of our page. It is time to submit it for reviewing.

First we have to commit our modifications with Git

```
git commit -am "Tutorial: How to contribute to PyMoDAQ documentation. Initial commit."
```

Note: If we also included some new files in the repository, like images, we have to tell Git to take those files under its supervision, which is done with the `git add -i` command. A simple command line interface will guide you to [select the files to add](#).

We then push our changes to our remote repository

```
git push
```

Finally, we will open a pull request to the upstream repository from the GitHub interface. Be careful to select the *pymodaq-dev* branch!

Those steps are explained with more details in the *plugin development tutorial*.

Fig. 7.170: Pull request to the upstream repository. Be careful to select the **pymodag-dev** branch!

Let's hope we will convince the owner that our tutorial is usefull! Thanks for contributing ;)

7.5.6 Updating your instrument plugin for PyMoDAQ 4

What's new in PyMoDAQ 4

The main modification in PyMoDAQ 4 concerning the instrument *plugins* is related to the hierarchy of the *modules* in the source code, see *What's new in PyMoDAQ 4*.

What should be modified

Imports

Mostly the only things to be modified are imports that should reflect the new package layout. This includes import in obvious files, for instance imports in the DAQ_Move_template plugin, see Fig. 7.171.

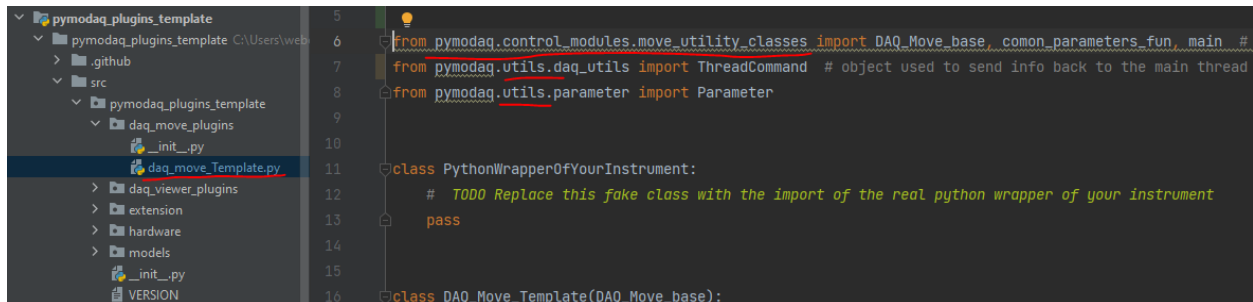


Fig. 7.171: New imports

Some imports are a bit more insidious. Indeed, often there is no specific code in the `__init__.py` files we see everywhere in our modules. But in the plugins, there is a bit of initialization code, see for instance Fig. 7.172 so make sure you changed the imports in all these `__init__.py`.

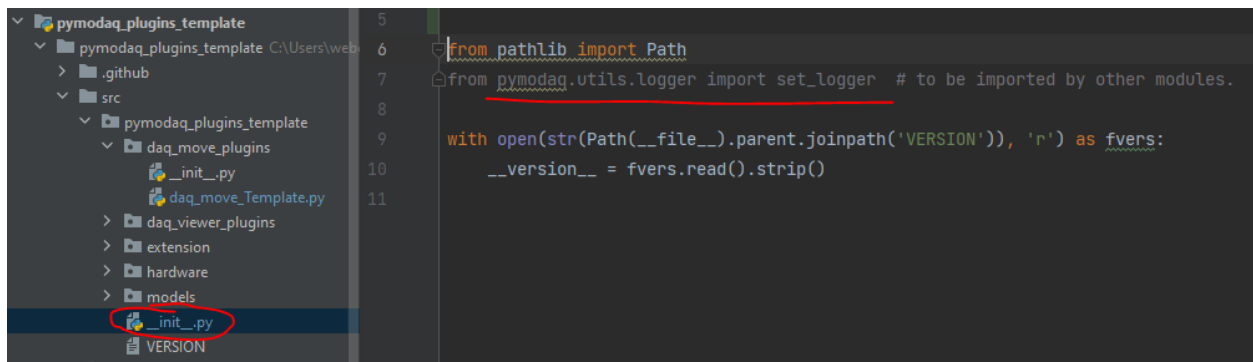


Fig. 7.172: New imports hidden in the `__init__.py` files

And that's it, they should be working now!

Note: If your instrument plugin has been written from a recent version of the template (say early 2023) then the only `__init__.py` file to be modified is the one in figure Fig. 7.172 but otherwise you'll need to modify most of them... sorry :-)

Data emission

But to make things very neat, your detector instrument plugins should emit no more lists of `DataFromPlugins` objects but a `DataToExport` emitted using new signals, see [Emission of data](#).

Requirements

And for the final bit, make sure to add a dependency to `pymodaq >= 4.0.0` in the package requirements, see Fig. 7.173. With this, the Plugin Manager will know your plugin is compatible and will propose it to installation.

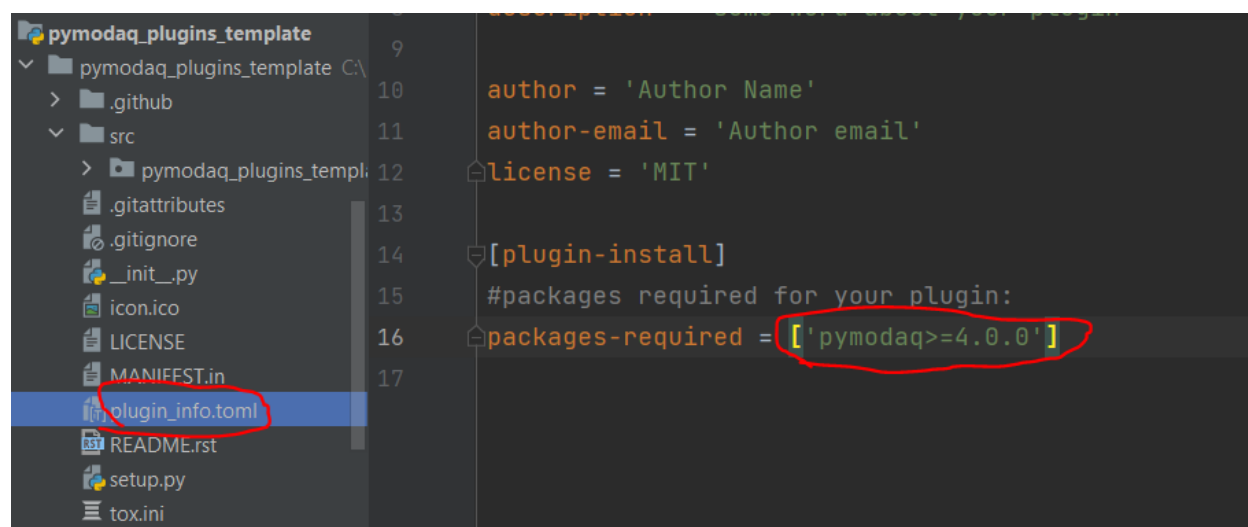


Fig. 7.173: Requirements necessary so that the Plugin Manager know your plugin is compatible with PyMoDAQ 4.

7.5.7 Tutorial On Data Manipulation and analysis

Author email	sebastien.weber@cemes.fr
Last update	february 2024
Difficulty	Intermediate

This tutorial is directly extracted from a jupyter notebook used to illustrate how to get your data back from a PyMoDAQ h5 file, analyze it and plot interactively these data. You'll find the notebook [here](#).

This example is using experimental data collected on a time-resolved optical spectroscopy set-up developed by Arnaud Arbouet and “PyMoDAQed” by Sebastien Weber in CEMES.

Practical work sessions exploiting this set-up and data analysis are organized every year in the framework of the Master PFIQMC at University Toulouse III PauL Sabatier. The students have to align an ultrafast transient absorption experiment and acquire data from a gold thin film. In these pump-probe experiments, two femtosecond collinear light pulses

are focused on the sample (see Fig. 7.174). The absorption by a first “pump” pulse places the sample out-of equilibrium. A second, delayed “probe” light pulse is used to measure the transmission of the sample during its relaxation.

The measured dynamics shows (i) the transmission change associated with the injection of energy by the pump pulse ($< \text{ps}$ timescale) followed by (ii) the quick thermalization of the electron gas with the gold film phonons (ps timescale) and (iii) the oscillations induced by the mechanical vibrations of the film (10s ps timescale). To be able to detect these oscillations, one needs to repeat the pump-probe scan many times and average the data.

PyMoDAQ allows this using the DAQ_Scan extension. One can specify how many scan should be performed and both the current scan and the averaged one are displayed live. However all the individual scans are saved as a multi-dimensional array. Moreover, because of the different time-scales (for electrons and for phonons) a “Sparse” 1D scan is used. It allows to quickly specify actuator values to be scanned in pieces (in the form of multiple *start:step:stop*). For instance scanning the electronic time window using a low step value and the phonon time window with a higher time step. The scan is therefore perfectly sampled but the time needed for one scan is reduced.

The author thanks Dr Arnaud Arbouet for the data and explanations. And if you don’t understand (or don’t care about) the physics, it’s not an issue as this notebook is here to show you how to load, manipulate and easily plot your data.

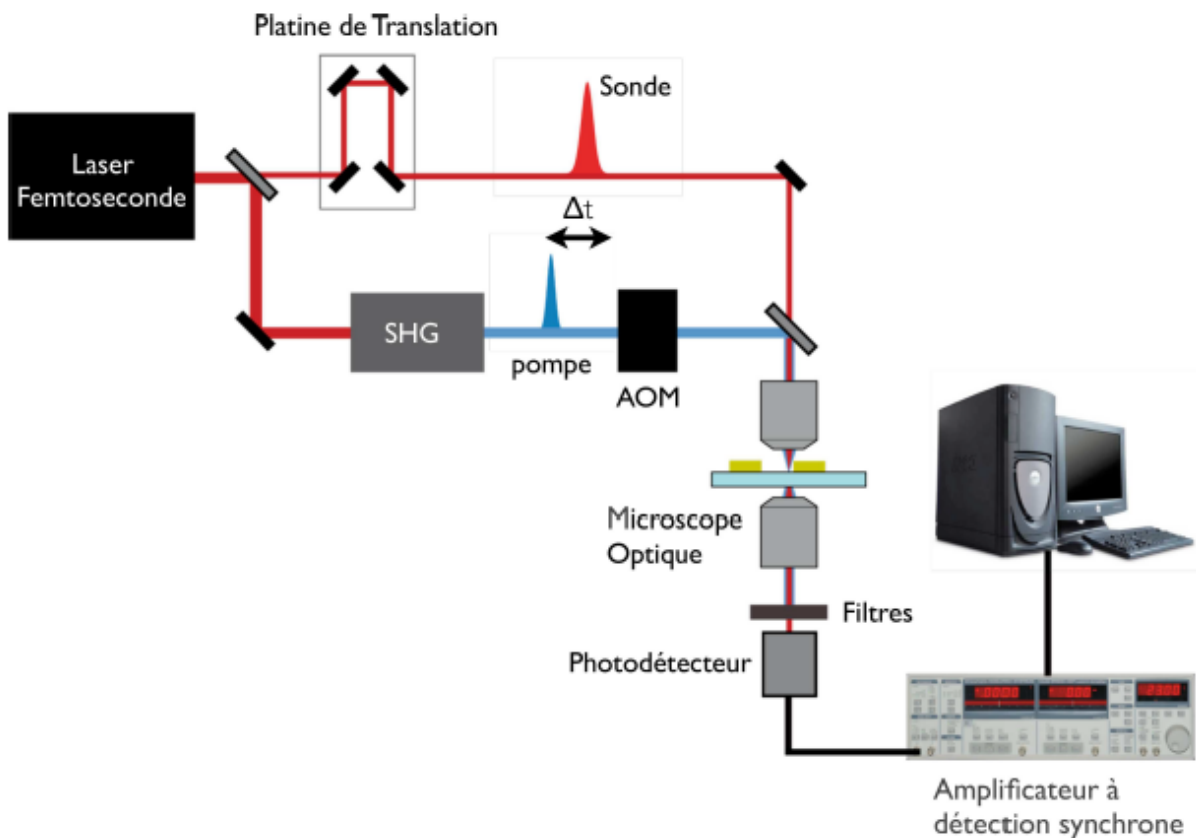


Fig. 7.174: Experimental Setup for time-resolved optical spectroscopy

To execute this tutorial properly, you’ll need PyMoDAQ $\geq 4.0.2$ (if not released yet, you can get it from github)

```
%gui qt5
# magic keyword only used to start a qt event loop within the jupyter notebook framework
```

(continues on next page)

(continued from previous page)

```

# importing built in modules
from pathlib import Path
import sys

# importing third party modules
import scipy as sc
import scipy.optimize as opt
import scipy.constants as cst
import numpy as np

# importing PymoDAQ modules
from pymodaq.utils.h5modules.saving import H5SaverLowLevel # object to open the h5 file
from pymodaq.utils.h5modules.data_saving import DataLoader # object used to properly
↳ load data from the h5 file
from pymodaq.utils.data import DataRaw, DataToExport

from pymodaq import __version__
print(__version__)

LIGHT_SPEED = 3e8 #m/s

```

4.2.0

Loading Data

```

dwa_loader = DataLoader('Dataset_20240206_000.h5') # this way of loading data directly
↳ from a Path is
#available from pymodaq>=4.2.0

for node in dwa_loader.walk_nodes():
    if 'Scan012' in str(node):
        print(node)

```

```

/RawData/Scan012 (GROUP) 'DAQScan'
/RawData/Scan012/Actuator000 (GROUP) 'delay'
/RawData/Scan012/Detector000 (GROUP) 'Lockin'
/RawData/Scan012/NavAxes (GROUP) ''
/RawData/Scan012/Detector000/Data0D (GROUP) ''
/RawData/Scan012/NavAxes/Axis00 (CARRAY) 'delay'
/RawData/Scan012/NavAxes/Axis01 (CARRAY) 'Average'
/RawData/Scan012/Detector000/Data0D/CH00 (GROUP) 'MAG'
/RawData/Scan012/Detector000/Data0D/CH01 (GROUP) 'PHA'
/RawData/Scan012/Detector000/Data0D/CH00/Data00 (CARRAY) 'MAG'
/RawData/Scan012/Detector000/Data0D/CH01/Data00 (CARRAY) 'PHA'

```

To load a particular node, use the `load_data` method

```

dwa_loaded = dwa_loader.load_data('/RawData/Scan012/Detector000/Data0D/CH00/Data00')
print(dwa_loaded)

```

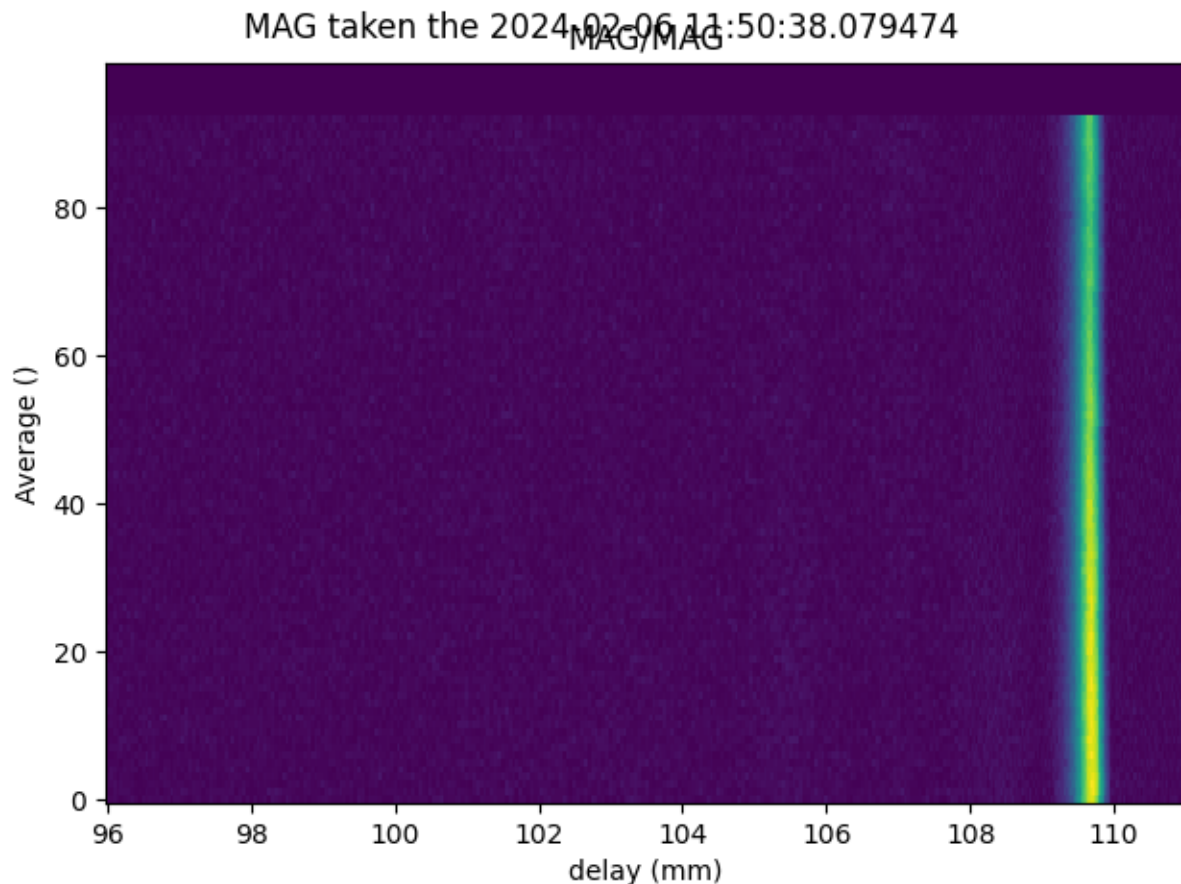


```
<DataWithAxes: MAG <len:1> (100, 392|)>
```

Plotting data

From PyMoDAQ 4.0.2 onwards, both the DataWithAxes (and its inheriting children classes) and the DataToExport objects have a plot method. One can specify as argument which backend to be used for plotting. At least two are available: matplotlib and qt. See below

```
dwa_loaded.nav_indexes = () # this is converting both navigation axes: average and
↪delay as signal axes (to be plotted in the Viewer2D)
dwa_loaded.plot('matplotlib')
```



or using PyMoDAQ's data viewer (interactive and with ROIs and all other features)

```
dwa_loaded.plot('qt')
```

The horizontal axis is a delay in millimeter (linear stage displacement, see setup) and we used a Sparsed scan with a non equal scan step (see figure below, right panel)

```
delay_axis = dwa_loaded.get_axis_from_index(1)[0]
dte = dwa_loaded.as_dte('mydata')
dte.append(DataRow(delay_axis.label, data=[delay_axis.get_data()]))
dte.plot('qt')
```

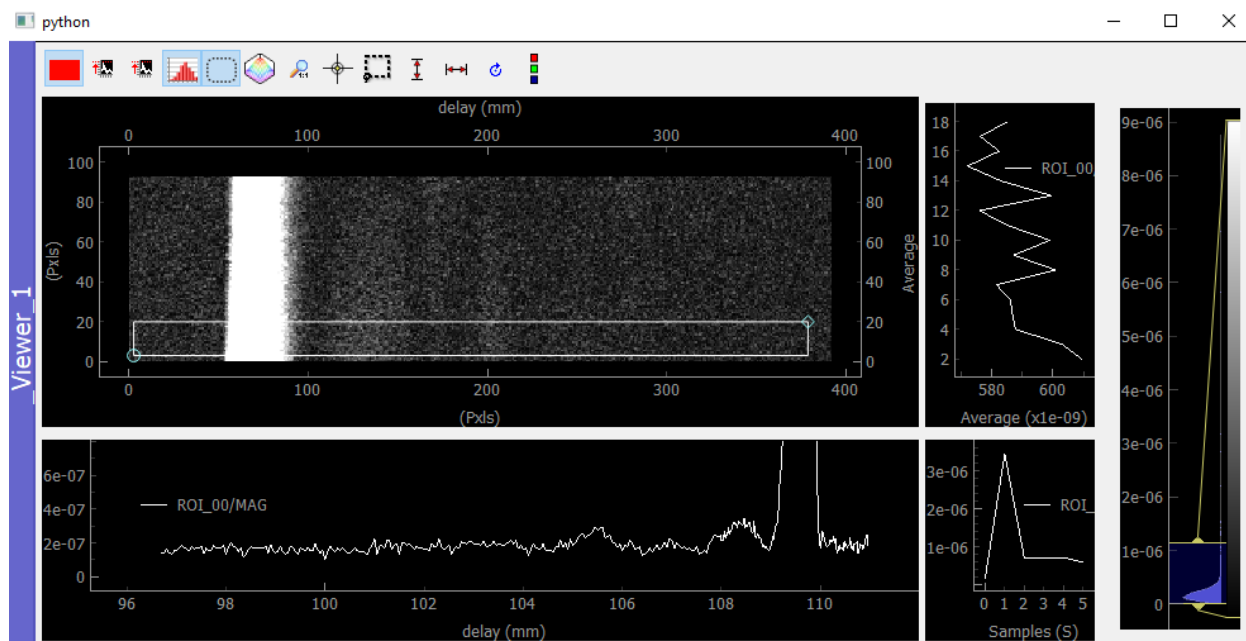


Fig. 7.175: python

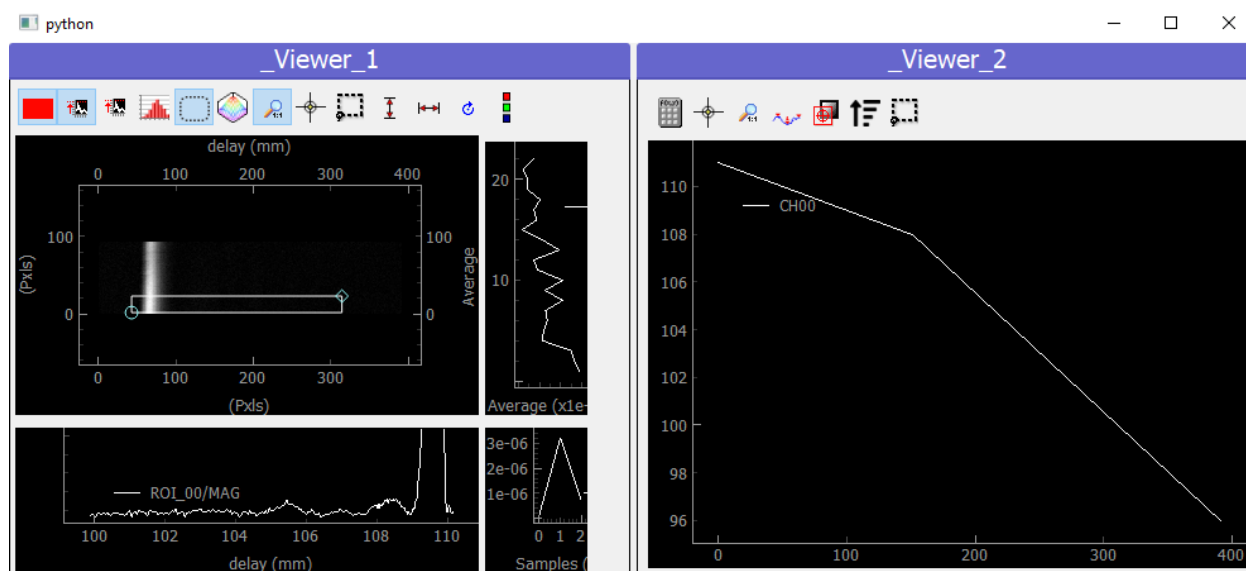


Fig. 7.176: python

```
dwa_loaded_steps = dwa_loaded.deepcopy()
delay_axis = dwa_loaded_steps.get_axis_from_index(1)[0]
delay_axis.data = delay_axis.create_simple_linear_data(len(delay_axis))
delay_axis.label = 'steps'
delay_axis.units = ''
```

This delay axis is for the moment in mm and reversed (the stage is going backwards to increase the delay). Let's recreate a flipped axis with seconds as units.

```
dwa_loaded_fs = dwa_loaded.deepcopy()
delay_axis = dwa_loaded_fs.get_axis_from_index(1)[0]
delay_axis.data = - 2 * delay_axis.get_data() / 1000 / LIGHT_SPEED # /1000 because the_
↳ displacement unit
# of the stage is in mm and the speed of light in m/s
delay_axis.data -= delay_axis.get_data()[0]
delay_axis.units = 's'
print(delay_axis.get_data()[0:10])
```

```
[0.000000000e+00 1.33333333e-13 2.66666667e-13 4.00000000e-13
 5.33333333e-13 6.66666667e-13 8.00000000e-13 9.33333333e-13
 1.06666667e-12 1.20000000e-12]
```

```
dwa_loaded_fs.plot('qt')
```

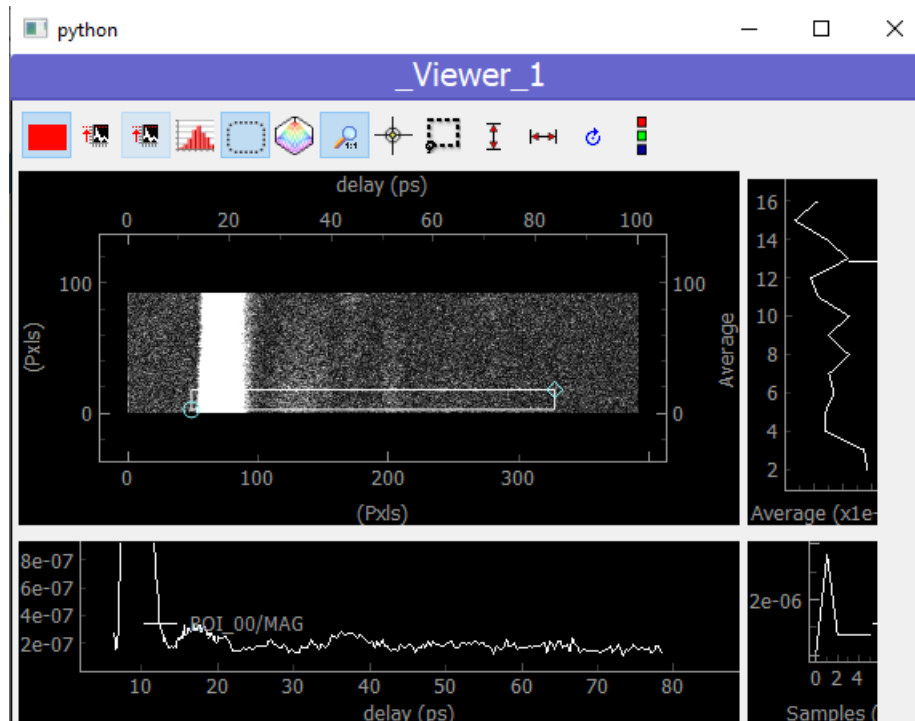


Fig. 7.177: python

Data Analysis

Now we got our data, one can extract infos from it

- life-time of the electrons -> phonons thermalization
- Oscillation period of the phonons vibration

To do this, one will properly slice the data corresponding to the electrons and the one corresponding to the phonons. To get the scan index to use for slicing, one will plot the raw data as a function of scan steps and extract the index using ROIs

```
dwa_loaded_steps.plot('qt')
```

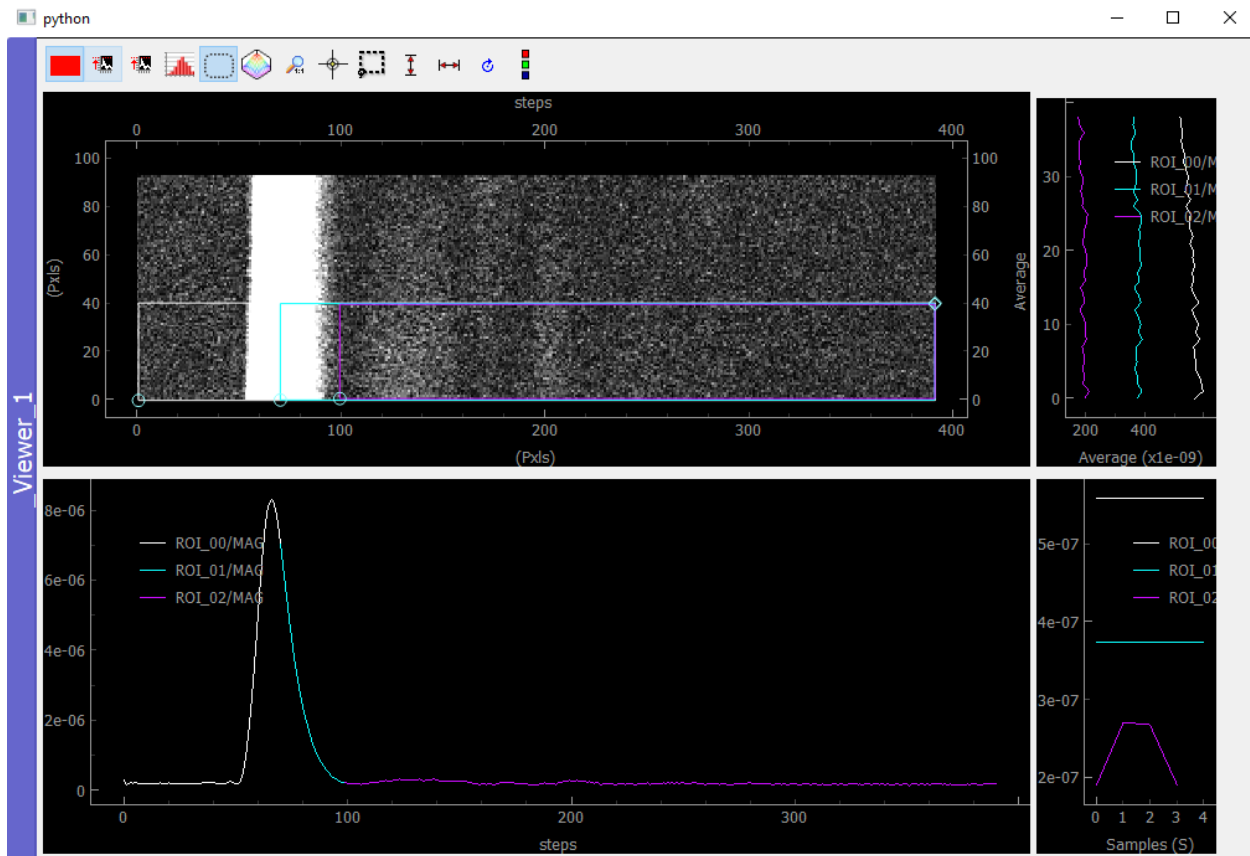


Fig. 7.178: python

Slicing the data

The ROi Manager (on the right, not visible here) tell us to use:

```
indexes_electrons = (70, 390)
indexes_phonons = (100, 300)
indexes_average = (0, 40) # we are not using all the averaging because the gold
# film seems to be dying as time goes on...
```

First we slice the data over the average indexes and the electron indexes. This is done easily using the `isig` slicer (sig for signal axes. For navigation one should use the `inav` slicer). Those slicers return a `DataWithAxes` object where data and axes have been sliced. Then we *immediately* apply the mean method over the average axis (index 0) to get 1D dimensionality data:

```
dwa_electrons = dwa_loaded_fs.isig[slice(*indexes_average), slice(*indexes_electrons)].
    ↪mean(0)
print(dwa_electrons)

dwa_phonons = dwa_loaded_fs.isig[slice(*indexes_average), slice(*indexes_phonons)].
    ↪mean(0)
print(dwa_phonons)
```

```
<DataWithAxes: MAG <len:1> (|320)>
<DataWithAxes: MAG <len:1> (|200)>
```

```
dte = DataToExport('mydata', data=[dwa_electrons, dwa_phonons])
print(dte)
dte.plot('qt')
```

```
DataToExport: mydata <len:2>
* <DataWithAxes: MAG <len:1> (|320)>
* <DataWithAxes: MAG <len:1> (|200)>
```

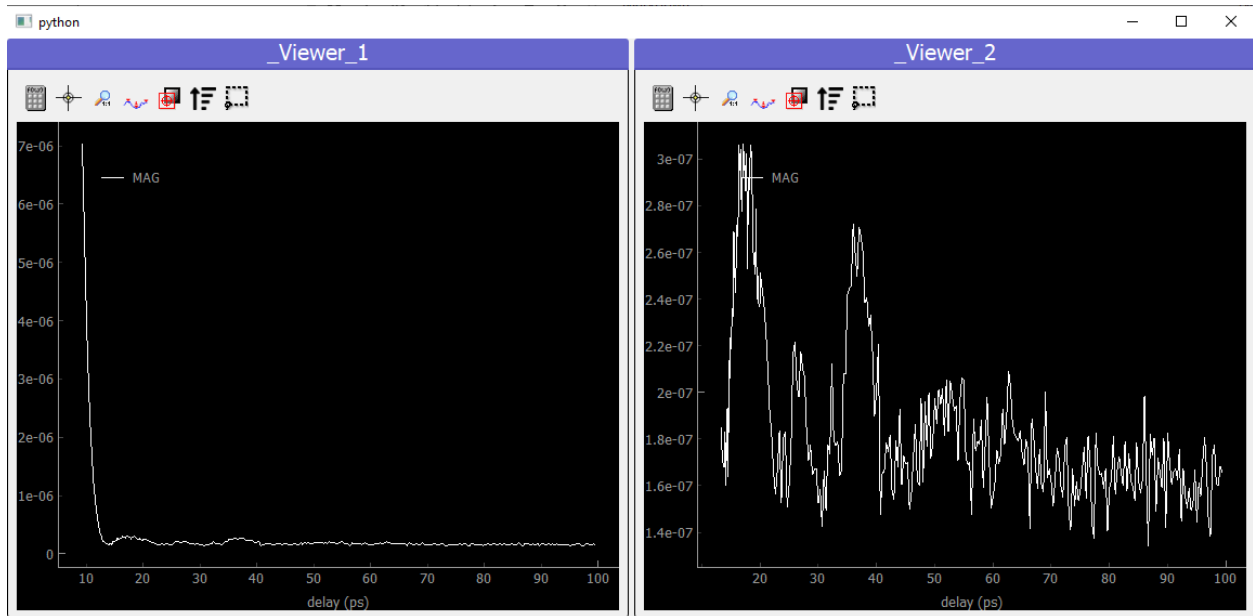


Fig. 7.179: python

Fitting the Data

Electrons:

```
def my_lifetime(x, A, B, C, tau):
    return A + C * np.exp(-(x - B)/tau)

time_axis = dwa_electrons.axes[0].get_data()
initial_guess = (2e-7, 10e-12, 7e-6, 3e-11)

dwa_electrons_fitted = dwa_electrons.fit(my_lifetime, initial_guess=initial_guess)
dwa_electrons_fitted.append(dwa_electrons)
dwa_electrons_fitted.plot('qt')
```

```
<pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D at 0x2ae0556cb80>
```

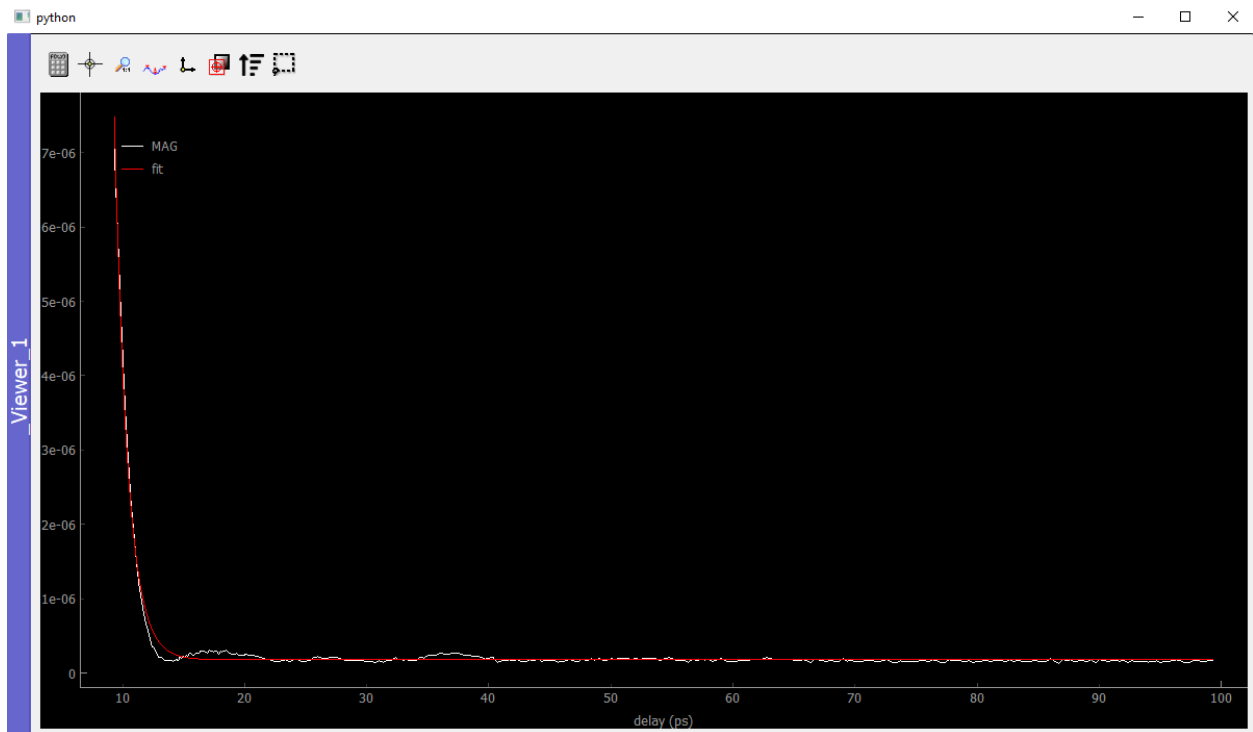


Fig. 7.180: python

One get a life time of about:

```
f'Life time: {dwa_electrons_fitted.fit_coeffs[0][3] * 1e12} ps'
```

```
'Life time: 1.0688184683663233 ps'
```

Phonons:

For the phonons, it seems we have to analyse oscillations. The best for this is a Fourier Transform analysis. However because of the sparse scan the sampling at the beginning is different from the one at the end. We'll have to resample our data on a regular grid before doing Fourier Transform

Resampling

```
from pymodaq.utils import math_utils as mutils
from pymodaq.utils.data import Axis
phonon_axis_array = dwa_phonons.get_axis_from_index(0)[0].get_data()
phonon_axis_array -= phonon_axis_array[0]
time_step = phonon_axis_array[-1] - phonon_axis_array[-2]
time_array_linear = mutils.linspace_step(0, phonon_axis_array[-1], time_step)
dwa_phonons_interp = dwa_phonons.interp(time_array_linear)

dwa_phonons_interp.plot('qt')
```

FFT

```
dwa_fft = dwa_phonons_interp.ft()

dwa_phonons_fft = DataToExport('FFT', data=[
    dwa_phonons_interp,
    dwa_fft.abs(),
    dwa_fft.abs(),
    dwa_fft.abs()])
dwa_phonons_fft.plot('qt')
```

Using advanced math processors to extract data from dwa:

```
from pymodaq.post_treatment.process_to_scalar import DataProcessorFactory
data_processors = DataProcessorFactory()
print('Implemented possible processing methods, can be applied to any data type and_
↳ dimensionality')
print(data_processors.keys)
dwa_processed = data_processors.get('argmax').process(dwa_fft.abs())
print(dwa_processed[0])
```

```
Implemented possible processing methods, can be applied to any data type and_
↳ dimensionality
['argmax', 'argmean', 'argmin', 'argstd', 'max', 'mean', 'min', 'std', 'sum']
[0.]
```

or using builtin math methods applicable only to 1D data:

```
dte_peaks = dwa_fft.abs().find_peaks(height=1e-6)
print(dte_peaks[0].axes[0].get_data() / (2*np.pi))

dte_peaks[0].axes[0].as_dwa().plot('matplotlib', 'o-r') # transforms an Axis object to_
```

(continues on next page)

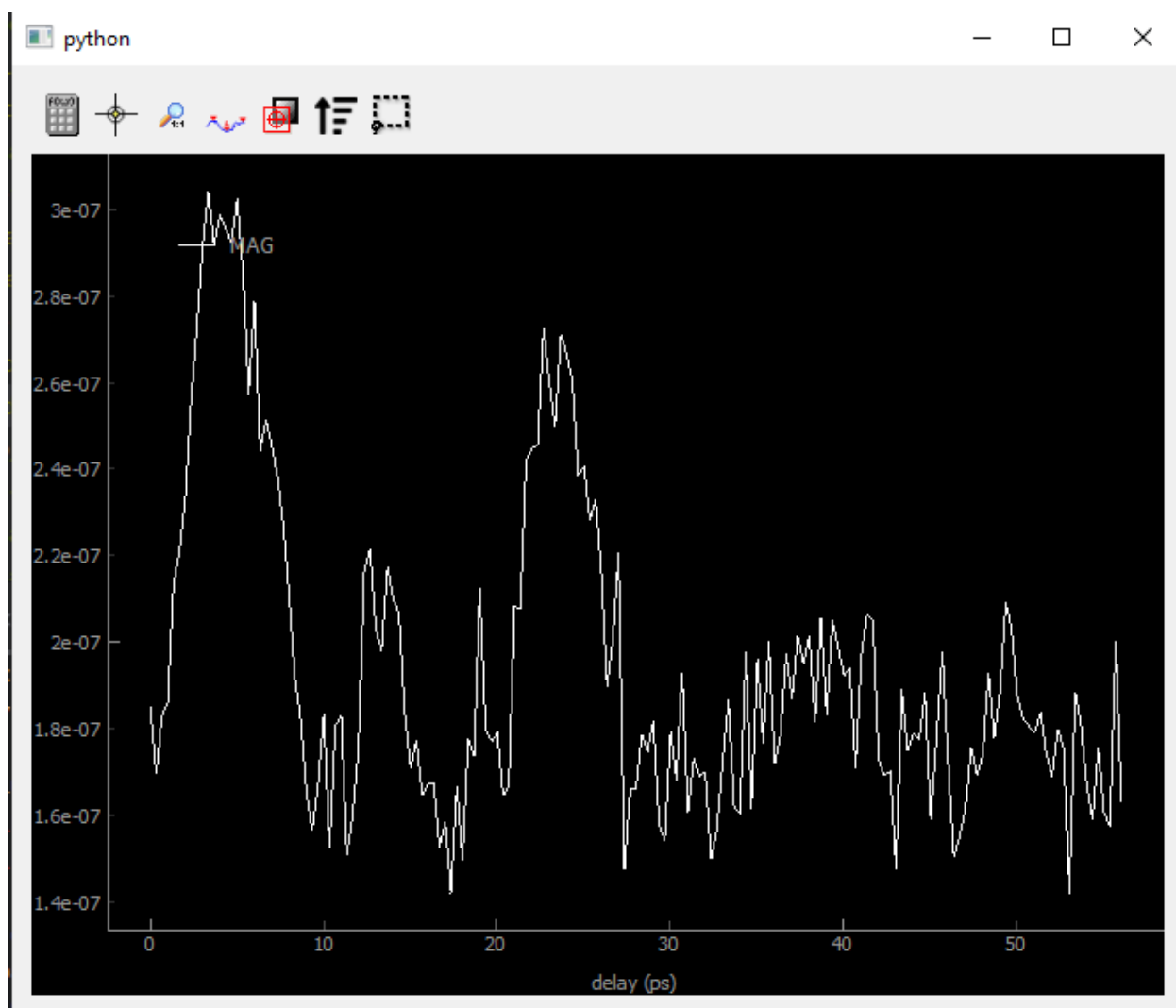


Fig. 7.181: Interpolated data on a regular time axis

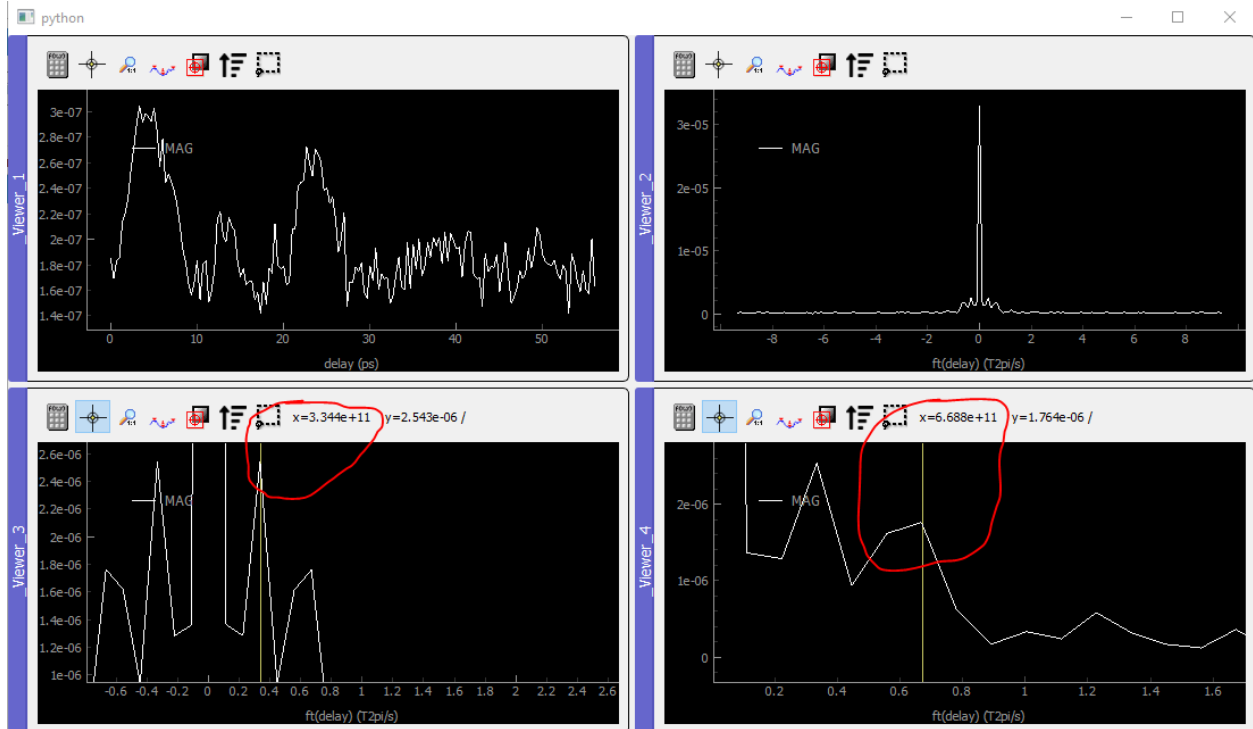


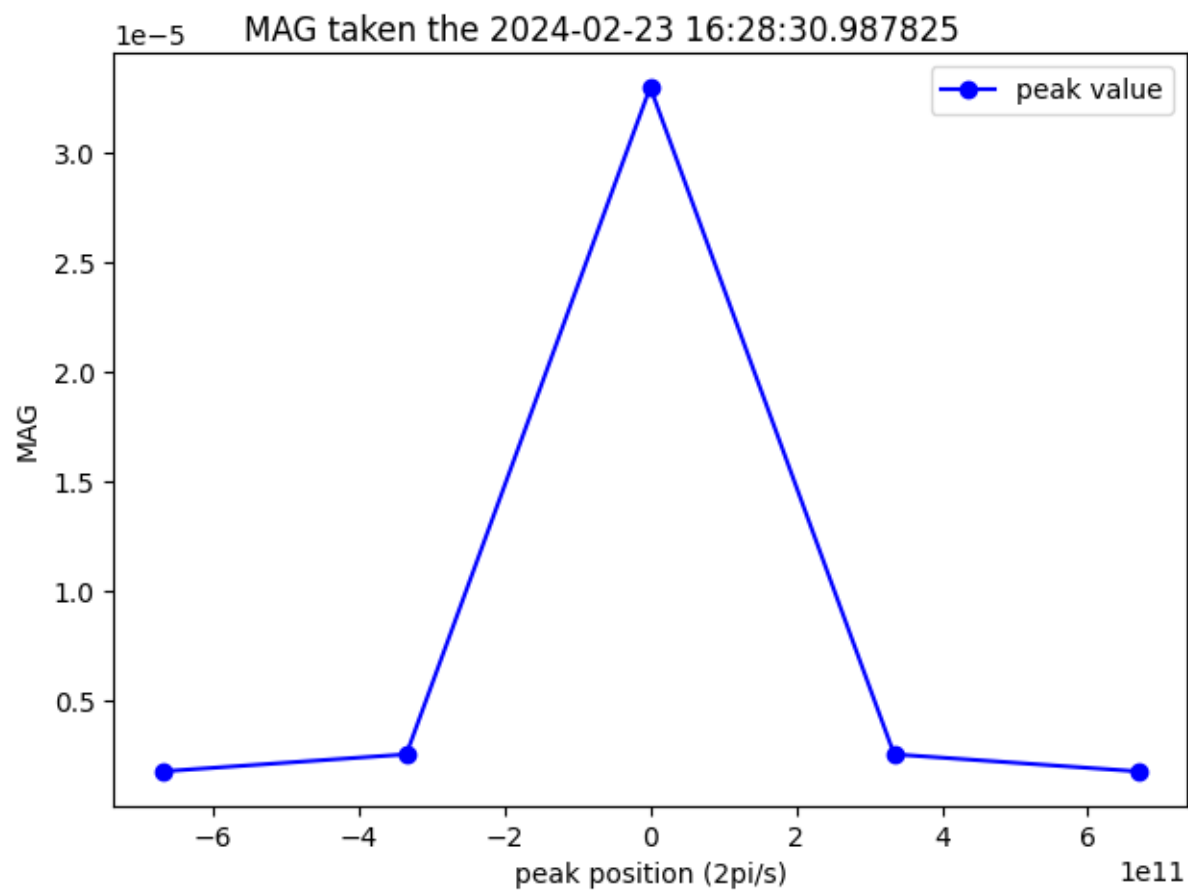
Fig. 7.182: Temporal data and FFT amplitude (top). Zoom over the two first harmonics (bottom)

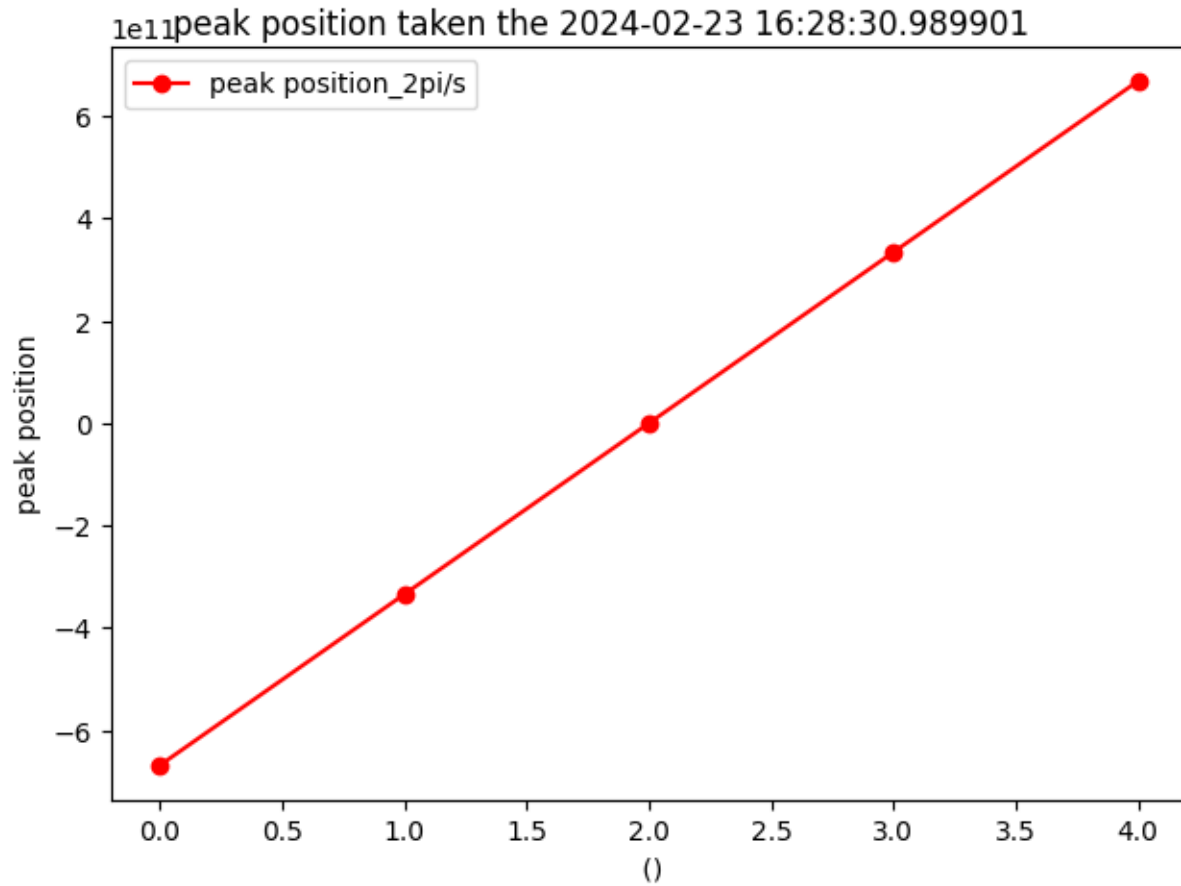
(continued from previous page)

↪ *dwa for quick plotting*

```
dte_peaks[0].get_data_as_dwa(0).plot('matplotlib', 'o-b') # select part of the data_
↪ object for "selected" plotting
```

```
[ -1.06435192e+11 -5.32175961e+10  0.00000000e+00  5.32175961e+10
  1.06435192e+11 ]
```





From this one get a fundamental frequency of $5.32e10$ Hz that corresponds to a period of:

```
T_phonons = 1/5.32e10
print(f'Period T = {T_phonons * 1e12} ps')
```

```
Period T = 18.796992481203006 ps
```

From this period and the speed of sound in gold, one can infer the gold film thickness:

```
thickness = T_phonons / 2 * SOUND_SPEED_GOLD
print(f"Gold Thickness: {thickness * 1e9} nm")
```

```
Gold Thickness: 30.45112781954887 nm
```

Summary

To summarize this tutorial, we learned to:

- easily load data using the *DataLoader* object and its *load_data* method (also using the convenience *walk_nodes* method to print all nodes from a file)
- easily plot loaded data using the *plot* method (together with the adapted backend)
- manipulate the data using its axes, navigation indexes, slicers and built in mathematical methods such as *mean*, 'abs', Fourier transforms, interpolation, fit...

For more details, see [Data Management](#)

7.6 Who use it?

- PyMoDAQ is used as the core acquisition program of several experiments at CEMES/CNRS and the main interface of its HC-IUMI Ultrafast Electron Microscope
- The attolab platform at CEA Saclay started using it in 2019

7.6.1 Institutions using PyMoDAQ



7.6.2 What they think of PyMoDAQ?

- *“The use of PyMoDAQ has really accelerated our experimental development by allowing to develop a modular acquisition system involving very different motorized stages or piezoactuators. It is now running everyday on our experiments, 100% reliable”*, Dr Arnaud Arbouet, Senior Researcher CEMES/CNRS
- *Pymodaq is a python framework for data acquisition. If your specific device driver is not yet implemented, that is the only thing you will have to do. Pymodaq take care of the rest. Graphical user interface, synchronization of the instruments and so on, is already implemented. Once you have implemented your driver, you can release it for the community. That is how Pymodaq will get more and more complete. Of course you need to invest a bit of your time to get used to it, but it is worth it!*, Dr David Bresteau, Researcher at CEA Saclay, Attolab platform.
- We are setting up Pymodaq on our scanning NV microscopy and deep UV spectroscopy experiments and we appreciate a lot its easy installation, its modularity and the automatic generation of the graphical interfaces, as well as the strong community support. The updates of the modules, the training sessions organized regularly and the numerous video tutorials also reflect the vitality of the community. We also start contributing by adding our own instruments and functionalities to share them with all the users. We fully support this great project! A. Finco, P. Valvin - L2C/S2QT

Note: If you are using PyMoDAQ and would like to help to promote the project, please send your feedback to bastien.weber@cemes.fr and we will include your message or logo on this page. If you wish to contribute, see [Contributing](#).

Note: If you wish to communicate with users of PyMoDAQ, a mailing list exists: pymodaq@services.cnrs.fr

7.6.3 Some Scientific publication on/using PyMoDAQ

- Weber, S. J. PyMoDAQ: An open-source Python-based software for modular data acquisition. [Review of Scientific Instruments](#), 92(4), 045104 (2021).
- Luttmann, M. *et al.* In Situ Sub-50-Attosecond Active Stabilization of the Delay Between Infrared and Extreme-Ultraviolet Light Pulses. [Physical Review Applied](#), 15(3), 034036 (2021).
- S. Meuret *et al.* Time-resolved cathodoluminescence in an ultrafast transmission electron microscope [Appl. Phys. Lett.](#) 119, 062106 (2021)
- F. Houdellier *et al.* Development of a high brightness ultrafast Transmission Electron Microscope based on a laser-driven cold field emission source [Ultramicroscopy](#), 186, 128 (2018).
- D. Bresteau *et al.* FAB10: a user-oriented bandwidth-tunable extreme ultraviolet lightsource for investigations of femtosecond to attosecond dynamics in gas and condensed phases [Eur. Phys. J. Spec. Top.](#) (2023)

7.7 Glossary Terms

Here are some definitions of the specific terms used in the PyMoDAQ documentation:

Actuator

Any instrument with a controllable varying parameter

Detector

Any instrument generating data to be recorded

Control Modules

GUI for actuators and detectors, with subsequent classes: `DAQ_Move` and `DAQ_Viewer`, see [Control Modules](#)

DashBoard

GUI allowing configuration and loading of a *preset* of actuators and detectors. You can also start extensions from its GUI such as the [DAQ Scan](#), [DAQ Logger](#), ... See [DashBoard](#)

Preset

XML file containing the number and type of control modules to be used for a given experiment. You can create, modify and load a preset from the Dashboard

DataSource

Enum informing about the source of the data object, for instance raw from a detector or processed from mathematical functions (from ROI, ...)

DataDim

Enum for the dimensionality representation of the data object, for instance scalars have a dimensionality *Data0D*, waveforms or vectors have *Data1D* dimensionality, camera's data are *Data2D*, and hyperspectral (or other) are *DataND*

DataDistribution

Enum for the distribution type of the data object. Data can be stored on linear grid (think about an oscilloscope trace having a fixed time interval, or camera having a regular grid of pixels) or stored on non uniform and non linear “positions”, for instance data taken at random time intervals. Data can therefore have two distributions: **uniform** or **spread**.

Signal

Signal and Navigation is a term taken from the hyperspy package vocabulary. It is useful when dealing with multidimensional data. Imagine data you obtained from a camera (256x1024 pixels) during a linear 1D scan of one actuator (100 steps). The final shape of the data would be (100, 256, 1024). The first dimension corresponds to a Navigation axis (the scan), and the rest to Signal axes (the real detector’s data). The corresponding data has a dimensionality of DataND and a representation of (100|256,1024).

Navigation

See above.

dwa

Short name for DataWithAxes object

dte

Short name for DataToExport object

Plugin

A plugin is a python package whose name is of the type: *pymodaq_plugins_apluginname* containing functionalities to be added to PyMoDAQ

Note: A plugin may contains added functionalities such as:

- **Classes to add a given instrument:** allows a given instrument to be added programmatically in a *Control Modules* graphical interface
- **Instrument drivers** located in a *hardware* folder: contains scripts/classes to ease communication with the instrument. Could be third party packages such as Pymeasure
- **PID models** located in a *models* folder: scripts and classes defining the behaviour of a given PID loop including several actuators or detectors, see *The PID Model*
- **Extensions** located in a *extensions* folder: scripts and classes allowing to build extensions on top of the *Dashboard*

Entry points python mechanism is used to let know PyMoDAQ of installed Instrument, PID models or extensions plugins

Module

A module in the python sense is an importable object either a directory containing an *__init__.py* file or a python file containing data, functions or classes.

Note: If there is code that can be executed within your module but you don’t want it to be executed when importing, make sure to protect the execution using a : `if __name__ == '__main__':` clause.

7.8 Library Reference

7.8.1 Control modules

ControlModule base classes

Both DAQ_Move and DAQ_Viewer control modules share some specificities and inherit from a base class: the *ControlModule*

The same is also true for the UI of these modules sharing a common UI base class: the *ControlModuleUI*

Summary of the classes dealing with the DAQ_Viewer control module:

DAQ_Viewer class

This documentation highlights the useful entry and output points that you may use in your applications.

DAQ_Detector class

The Detector class is an object leaving in the plugin thread and responsible for the communication between DAQ_Viewer and the plugin itself

The Viewer UI class

This object is the User Interface of the DAQ_Viewer, allowing easy access to all of the DAQ_Viewer fonctionnalités in a generic interface.

`pymodaq.control_modules.
move_utility_classes.params`

Built-in mutable sequence.

The DAQ_Move Class

This documentation highlights the useful entry and output points that you may use in your applications.

The DAQ_Move UI class

This object is the User Interface of the DAQ_Viewer, allowing easy access to all of the DAQ_Viewer fonctionnalités in a generic interface.

The DAQ_Move Plugin Class

This object is the base class from which all actuator plugins should inherit. It exposes a few methods, attributes and signal that could be useful to understand.

```
class pymodaq.control_modules.move_utility_classes.DAQ_Move_base(parent: Optional[DAQ_Move_Hardware]  
                                                                = None, params_state:  
                                                                Optional[dict] = None)
```

The base class to be inherited by all actuator modules

This base class implements all necessary parameters and methods for the plugin to communicate with its parent (the DAQ_Move module)

Parameters

- **parent** (*DAQ_Move_Hardware*) –
- **params_state** (*Parameter*) – pyqtgraph Parameter instance from which the module will get the initial settings (as defined in the preset)

move_done_signal

signal represented by a float. Is emitted each time the hardware reached the target position within the epsilon precision (see `comon_parameters` variable)

Type

Signal

controller

the object representing the hardware in the plugin. Used to access hardware functionality

Type

object

settings

instance representing the hardware settings defined from the params attribute. Modifications on the GUI settings

will be transferred to this attribute. It stores at all times the current state of the hardware/plugin settings

Type

Parameter

params

Its definition on the class level enable the automatic update of the GUI settings when changing plugins (even in managers mode creation). To be populated on the plugin level as the base class doesn't represent a real hardware

Type

List of dict used to create a Parameter object.

is_multiaxes

class level attribute. Defines if the plugin controller controls multiple axes. If True, one has to define a Master instance of this plugin and slave instances of this plugin (all sharing the same `controller_ID` parameter)

Type

bool

current_value

stores the current position after each call to the `get_actuator_value` in the plugin

Type

DataActuator

target_value

stores the target position the controller should reach within epsilon

Type

DataActuator

Attributes*axis_name*

Get/Set the current axis using its string identifier

axis_names

Get/Set the names of all axes controlled by this instrument plugin

axis_value

Get the current value selected from the current axis

controller_units

Get/Set the units of this plugin

current_position**current_value***ispolling*

Get/Set the polling status

target_position**target_value**

Methods

<i>check_bound</i> (position)	Check if the current position is within the software bounds
<i>commit_settings</i> (param)	to subclass to transfer parameters to hardware
<i>emit_status</i> (status)	Emit the status_sig signal with the given status ThreadCommand back to the main GUI.
<i>emit_value</i> (pos)	Convenience method to emit the current actuator value back to the UI
<i>get_position_with_scaling</i> (pos)	Get the current position from the hardware with scaling conversion.
<i>ini_attributes</i> ()	To be subclassed, in order to init specific attributes needed by the real implementation
<i>ini_stage_init</i> ([old_controller, new_controller])	Manage the Master/Slave controller issue
<i>move_done</i> ([position])	Emit a move done signal transmitting the float position to hardware.
<i>poll_moving</i> ()	Poll the current moving.
<i>send_param_status</i> (param, changes)	Send changes value updates to the gui to update consequently the User Interface
<i>set_position_relative_with_scaling</i> (pos)	Set the scaled positions in case of relative moves
<i>set_position_with_scaling</i> (pos)	Set the current position from the parameter and hardware with scaling conversion.
<i>update_settings</i> (settings_parameter_dict)	Receive the settings_parameter signal from the param_tree_changed method and make hardware updates of modified values.

check_target_reached	
commit_common_settings	
get_actuator_value	
move_abs	
move_done_signal	
move_home	
move_rel	

check_bound(*position*: DataActuator) → DataActuator

Check if the current position is within the software bounds

Return the new position eventually coerced within the bounds

commit_settings(*param*: Parameter)

to subclass to transfer parameters to hardware

emit_status(*status*: ThreadCommand)

Emit the status_sig signal with the given status ThreadCommand back to the main GUI.

emit_value(*pos*: DataActuator)

Convenience method to emit the current actuator value back to the UI

get_position_with_scaling(*pos*: DataActuator) → DataActuator

Get the current position from the hardware with scaling conversion.

ini_attributes()

To be subclassed, in order to init specific attributes needed by the real implementation

ini_stage_init(*old_controller=None, new_controller=None*)

Manage the Master/Slave controller issue

First initialize the status dictionary Then check whether this stage is controlled by a multiaxe controller (to be defined for each plugin)

if it is a multiaxes controller then: * if it is Master: init the controller here * if it is Slave: use an already initialized controller (defined in the preset of the dashboard)

Parameters

- **old_controller** (*object*) – The particular object that allow the communication with the hardware, in general a python wrapper around the hardware library. In case of Slave this one comes from a previously initialized plugin
- **new_controller** (*object*) – The particular object that allow the communication with the hardware, in general a python wrapper around the hardware library. In case of Master it is the new instance of your plugin controller

move_done(*position: Optional[DataActuator] = None*)

Emit a move done signal transmitting the float position to hardware.

The position argument is just there to match some signature of child classes.

Arguments	Type	Description
<i>position</i>	float	The position argument is just there to match some signature of child classes

poll_moving()

Poll the current moving. In case of timeout emit the raise timeout Thread command.

See also:

DAQ_utils.ThreadCommand, [move_done](#)

send_param_status(*param, changes*)

Send changes value updates to the gui to update consequently the User Interface

The message passing is made via the ThreadCommand “update_settings”.

set_position_relative_with_scaling(*pos: DataActuator*) → DataActuator

Set the scaled positions in case of relative moves

set_position_with_scaling(*pos: DataActuator*) → DataActuator

Set the current position from the parameter and hardware with scaling conversion.

update_settings(*settings_parameter_dict*)

Receive the settings_parameter signal from the param_tree_changed method and make hardware updates of modified values.

property axis_name: Union[str, object]

Get/Set the current axis using its string identifier

property axis_names: `Union[List, Dict]`

Get/Set the names of all axes controlled by this instrument plugin

Return type

List of string or dictionary mapping names to integers

property axis_value: `object`

Get the current value selected from the current axis

property controller_units

Get/Set the units of this plugin

property ispolling

Get/Set the polling status

7.8.2 Extensions

DAQ_Scan module

The Bayesian Extension and utilities

Summary of the main classes for the Bayesian Optimization extension

<i>BayesianOptimisation</i> (dockarea, dashboard)	PyMoDAQ extension of the DashBoard to perform the optimization of a target signal taken form the detectors as a function of one or more parameters controlled by the actuators.
<i>BayesianModelGeneric</i> (optimisation_controller)	
Methods	
<i>BayesianModelDefault</i> (optimisation_controller)	
Methods	

The Extension module

class `pymodaq.extensions.BayesianOptimisation`(*dockarea, dashboard*)

PyMoDAQ extension of the DashBoard to perform the optimization of a target signal taken form the detectors as a function of one or more parameters controlled by the actuators.

Attributes

modules_manager

useful tool to interact with DAQ_Moves and DAQ_Viewers

Methods

<code>connect_things()</code>	Connect actions and/or other widgets signal to methods
<code>setup_actions()</code>	Method where to create actions to be subclassed.
<code>setup_docks()</code>	to be subclassed to setup the docks layout for instance:
<code>setup_menu()</code>	to be subclassed create menu for actions contained into the self.actions_manager, for instance:
<code>value_changed(param)</code>	to be subclassed for actions to perform when one of the param's value in self.settings is changed

clean_h5_temp	
command_runner	
enable_controls_opti	
format_bounds	
get_set_model_params	
get_stopping_parameters	
go_to_best	
ini_live_plot	
ini_model	
ini_optimisation_runner	
ini_temp_file	
optimisation_done	
optimisation_done_signal	
process_output	
quit	
run_optimisation	
set_algorithm	
set_model	
update_actuators	
update_bounds	
update_data_plot	
update_stopping_criteria	
update_utility_function	

connect_things()

Connect actions and/or other widgets signal to methods

setup_actions()

Method where to create actions to be subclassed. Mandatory

Examples

```
>>> self.add_action('Quit', 'close2', "Quit program")
>>> self.add_action('Grab', 'camera', "Grab from camera", checkable=True)
>>> self.add_action('Load', 'Open', "Load target file (.h5, .png, .jpg) or_
↳data from camera", checkable=False)
>>> self.add_action('Save', 'SaveAs', "Save current data", checkable=False)
```

See also:

ActionManager.add_action

setup_docks()

to be subclassed to setup the docks layout for instance:

```
self.docks['ADock'] = gutils.Dock('ADock name) self.dockarea.addDock(self.docks['ADock'])
self.docks['AnotherDock'] = gutils.Dock('AnotherDock name) self.dockarea.addDock(self.docks['AnotherDock']),
'bottom', self.docks['ADock'])
```

See also:

pyqtgraph.dockarea.Dock

setup_menu()

to be subclassed create menu for actions contained into the self.actions_manager, for instance:

For instance:

```
file_menu = self.menubar.addMenu('File') self.actions_manager.affect_to('load', file_menu)
self.actions_manager.affect_to('save', file_menu)

file_menu.addSeparator() self.actions_manager.affect_to('quit', file_menu)
```

value_changed(param)

to be subclassed for actions to perform when one of the param's value in self.settings is changed

For instance: if param.name() == 'do_something':

```
if param.value():
    print('Do something') self.settings.child('main_settings', 'some-
thing_done').setValue(False)
```

Parameters

param ((Parameter) the parameter whose value just changed) –

property modules_manager: *ModulesManager*

useful tool to interact with DAQ_Moves and DAQ_Viewers

Will be available if a DashBoard has been set

Return type

ModulesManager

The Base Models

`class pymodaq.extensions.BayesianModelGeneric(optimisation_controller: BayesianOptimisation)`

Methods

<code>convert_input(measurements)</code>	Convert the measurements in the units to be fed to the Optimisation Controller :param measurements: data object exported from the detectors from which the model extract a float value (fitness) to be fed to the algorithm :type measurements: DataToExport
<code>convert_output(outputs[, best_individual])</code>	Convert the output of the Optimisation Controller in units to be fed into the actuators :param outputs: output value from the controller from which the model extract a value of the same units as the actuators :type outputs: list of numpy ndarray :param best_individual: the coordinates of the best individual so far :type best_individual: np.ndarray
<code>ini_model()</code>	To be subclassed
<code>runner_initialized()</code>	To be subclassed
<code>update_plots()</code>	Called when updating the live plots
<code>update_settings(param)</code>	Get a parameter instance whose value has been modified by a user on the UI To be overwritten in child class

check_modules	
ini_model_base	
optimisation_algorithm	
update_detector_names	

convert_input(*measurements*: DataToExport) → float

Convert the measurements in the units to be fed to the Optimisation Controller :param measurements: data object exported from the detectors from which the model extract a float value

(fitness) to be fed to the algorithm

Return type

float

convert_output(*outputs*: List[ndarray], *best_individual*=None) → DataToActuators

Convert the output of the Optimisation Controller in units to be fed into the actuators :param outputs: output value from the controller from which the model extract a value of the same units as the actuators :type outputs: list of numpy ndarray :param best_individual: the coordinates of the best individual so far :type best_individual: np.ndarray

Returns

DataToActuatorOpti – attribute, either ‘rel’ for relative or ‘abs’ for absolute.

Return type

derived from DataToExport. Contains value to be fed to the actuators with a a mode

ini_model()

To be subclassed

Initialize whatever is needed by your custom model

runner_initialized()

To be subclassed

Initialize whatever is needed by your custom model after the optimization runner is initialized

update_plots()

Called when updating the live plots

update_settings(param: Parameter)

Get a parameter instance whose value has been modified by a user on the UI To be overwritten in child class

class pymodaq.extensions.**BayesianModelDefault**(*optimisation_controller*: BayesianOptimisation)

Methods

<i>convert_input</i> (measurements)	Convert the measurements in the units to be fed to the Optimisation Controller
<i>convert_output</i> (outputs[, best_individual])	Convert the output of the Optimisation Controller in units to be fed into the actuators :param outputs: output value from the controller from which the model extract a value of the same units as the actuators :type outputs: list of numpy ndarray :param best_individual: the coordinates of the best individual so far :type best_individual: np.ndarray
<i>ini_model</i> ()	To be subclassed
<i>update_settings</i> (param)	Get a parameter instance whose value has been modified by a user on the UI To be overwritten in child class

optimize_from	
---------------	--

convert_input(*measurements*: DataToExport) → float

Convert the measurements in the units to be fed to the Optimisation Controller

Parameters

measurements ([DataToExport](#)) – data object exported from the detectors from which the model extract a float value (fitness) to be fed to the algorithm

Return type

float

convert_output(*outputs*: List[ndarray], *best_individual*=None) → DataToActuators

Convert the output of the Optimisation Controller in units to be fed into the actuators :param outputs: output value from the controller from which the model extract a value of the same units as the actuators :type outputs: list of numpy ndarray :param best_individual: the coordinates of the best individual so far :type best_individual: np.ndarray

Returns

- **DataToActuators** (*derived from DataToExport. Contains value to be fed to the actuators*)
- with a *mode* attribute, either 'rel' for relative or 'abs' for absolute.

ini_model()

To be subclassed

Initialize whatever is needed by your custom model

update_settings(param: Parameter)

Get a parameter instance whose value has been modified by a user on the UI To be overwritten in child class

The CustomApp base class

CustomApp(parent[, dashboard])

Base Class to ease the implementation of User Interfaces

```
class pymodaq.utils.gui_utils.CustomApp(parent: Union[DockArea, QWidget], dashboard: DashBoard = None)
```

Base Class to ease the implementation of User Interfaces

Inherits the MixIns ActionManager and ParameterManager classes. You have to subclass some methods and make concrete implementation of a given number of methods:

- setup_actions: mandatory, see [pymodaq.utils.managers.action_manager.ActionManager](#)
- value_changed: non mandatory, see [pymodaq.utils.managers.parameter_manager.ParameterManager](#)
- child_added: non mandatory, see [pymodaq.utils.managers.parameter_manager.ParameterManager](#)
- param_deleted: non mandatory, see [pymodaq.utils.managers.parameter_manager.ParameterManager](#)
- setup_docks: mandatory
- setup_menu: non mandatory
- connect_things: mandatory

Parameters

- **parent** (*DockArea or QWidget*) –
- **dashboard** (*DashBoard, optional*) –

See also:

[pymodaq.utils.managers.action_manager.ActionManager](#), [pymodaq.utils.managers.parameter_manager.ParameterManager](#), [pymodaq.utils.managers.modules_manager.ModulesManager](#), [pymodaq.dashboard.DashBoard](#)

Attributes

[modules_manager](#)

useful tool to interact with DAQ_Moves and DAQ_Viewers

Methods

<code>connect_things()</code>	Connect actions and/or other widgets signal to methods
<code>setup_docks()</code>	Mandatory method to be subclassed to setup the docks layout
<code>setup_menu()</code>	Non mandatory method to be subclassed in order to create a menubar

log_signal	
setup_ui	

connect_things()

Connect actions and/or other widgets signal to methods

setup_docks()

Mandatory method to be subclassed to setup the docks layout

Examples

```
>>>self.docks['ADock'] = gutils.Dock('ADock name') >>>self.dockarea.addDock(self.docks['ADock'])
>>>self.docks['AnotherDock'] = gutils.Dock('AnotherDock name') >>>self.dockarea.addDock(self.docks['AnotherDock'])
'bottom', self.docks['ADock'])
```

See also:

`pyqtgraph.dockarea.Dock`

setup_menu()

Non mandatory method to be subclassed in order to create a menubar

create menu for actions contained into the `self._actions`, for instance:

Examples

```
>>>file_menu = self._menubar.addMenu('File') >>>self.affect_to('load', file_menu)
>>>self.affect_to('save', file_menu)
>>>file_menu.addSeparator() >>>self.affect_to('quit', file_menu)
```

See also:

`pymodaq.utils.managers.action_manager.ActionManager`

property modules_manager: *ModulesManager*

useful tool to interact with `DAQ_Moves` and `DAQ_Viewers`

Will be available if a `DashBoard` has been set

Return type

ModulesManager

7.8.3 Utility Modules

Hdf5 module and classes

Hdf5 backends

The H5Backend is a wrapper around three hdf5 python packages: pytables, h5py and h5pyd. It allows seamless integration of any of these with PyMoDAQ features.

```
class pymodaq.utils.h5modules.backends.H5Backend(backend='tables')
```

Attributes

filename
h5file

Methods

<code>add_group(group_name, group_type, where[, ...])</code>	Add a node in the h5 file tree of the group type :param group_name: :type group_name: (str) a custom name for this group :param group_type: one of the possible values of GroupType :type group_type: str or GroupType enum :param where: :type where: (str or node) parent node where to create the new group :param metadata: :type metadata: (dict) extra metadata to be saved with this new group node
<code>close_file()</code>	Flush data and close the h5file
<code>create_earray(where, name, dtype[, ...])</code>	create enlargeable arrays from data with a given shape and of a given type.
<code>create_vlarray(where, name, dtype[, title])</code>	create variable data length and type and enlargeable 1D arrays
<code>define_compression(compression, compression_opts)</code>	Define cmpression library and level of compression :param compression: but zlib is used by pytables while gzip is used by h5py :type compression: (str) either gzip and zlib are supported here as they are compatible :param compression_opts (int): :type compression_opts (int): 0 to 9 0: None, 9: maximum compression
<code>get_children(where)</code>	Get a dict containing all children node hanging from where with their name as keys and types among Node, CARRAY, EARRAY, VLARRAY or StringARRAY
<code>get_node_name(node)</code>	return node name :param node (str or node instance): :param see h5py and pytables documentation on nodes:
<code>get_node_path(node)</code>	return node path :param node (str or node instance): :param see h5py and pytables documentation on nodes:
<code>get_set_group(where, name[, title])</code>	Retrieve or create (if absent) a node group Get attributed to the class attribute <code>current_group</code>
<code>is_node_in_group(where, name)</code>	Check if a given node with name is in the group defined by where (comparison on lower case strings) :param where: path or parent node instance :type where: (str or node) :param name: group node name :type name: (str)

create_carray	
flush	
get_attr	
get_group_by_title	
get_node	
get_parent_node	
has_attr	
isopen	
open_file	
read	
root	
save_file_as	
set_attr	
walk_groups	
walk_nodes	

add_group(*group_name*, *group_type*: *GroupType*, *where*, *title*="", *metadata*={}) → *GROUP*

Add a node in the h5 file tree of the group type :param *group_name*: :type *group_name*: (str) a custom name for this group :param *group_type*: one of the possible values of *GroupType* :type *group_type*: str or *GroupType* enum :param *where*: :type *where*: (str or node) parent node where to create the new group :param *metadata*: :type *metadata*: (dict) extra metadata to be saved with this new group node

Returns

(node)

Return type

newly created group node

close_file()

Flush data and close the h5file

create_earray(*where*, *name*, *dtype*, *data_shape*=None, *title*="")

create enlargeable arrays from data with a given shape and of a given type. The array is enlargeable along the first dimension

create_vlarray(*where*, *name*, *dtype*, *title*="")

create variable data length and type and enlargeable 1D arrays

Parameters

- **where** ((*str*) group location in the file where to create the array node) –
- **name** ((*str*) name of the array) –
- **dtype** ((*dtype*) numpy dtype style, for particular case of strings, use *dtype*='string') –
- **title** ((*str*) node title attribute (written in capitals)) –

Return type

array

define_compression(*compression*, *compression_opts*)

Define cmpression library and level of compression :param *compression*: but zlib is used by pytables while gzip is used by h5py :type *compression*: (str) either gzip and zlib are supported here as they are compatible :param *compression_opts* (int): :type *compression_opts* (int): 0 to 9 0: None, 9: maximum compression

get_children(*where*)

Get a dict containing all children node hanging from where with their name as keys and types among Node, CARRAY, EARRAY, VLARRAY or StringARRAY

Parameters

instance) (*where* (*str* or *node*) – see h5py and pytables documentation on nodes, and Node objects of this module

Returns

dict

Return type

keys are children node names, values are the children nodes

See also:

GROUP.children_name()

get_node_name(*node*)

return node name :param node (str or node instance): :param see h5py and pytables documentation on nodes:

Returns

str

Return type

name of the node

get_node_path(*node*)

return node path :param node (str or node instance): :param see h5py and pytables documentation on nodes:

Returns

str

Return type

full path of the node

get_set_group(*where, name, title=""*)

Retrieve or create (if absent) a node group Get attributed to the class attribute `current_group`

Parameters

- **where** (*str* or *node*) – path or parent node instance
- **name** (*str*) – group node name
- **title** (*str*) – node title

Returns

group

Return type

group node

is_node_in_group(*where, name*)

Check if a given node with name is in the group defined by where (comparison on lower case strings)
:param where: path or parent node instance :type where: (str or node) :param name: group node name
:type name: (str)

Returns

True if node exists, False otherwise

Return type

bool

Low Level saving

H5SaverBase and H5Saver classes are a help to save data in a hierarchical hdf5 binary file through the H5Backend object and allowing integration in the PyMoDAQ Framework. These objects allow the creation of a file, of the various nodes necessary to save PyMoDAQ's data. The saving functionalities are divided in two objects: *H5SaverBase* and *H5Saver*. *H5SaverBase* contains everything needed for saving, while *H5Saver*, inheriting *H5SaverBase*, adds Qt functionality such as emitted signals. However, these are not specific of PyMoDAQ's data types. To save and load data, one should use higher level objects, see [High Level saving/loading](#).

Created the 15/11/2022

@author: Sebastien Weber

```
class pymodaq.utils.h5modules.saving.H5Saver(*args, **kwargs)
```

status_sig: Signal

emits a signal of type Threadcommand in order to send log information to a main UI

new_file_sig: Signal

emits a boolean signal to let the program know when the user pressed the new file button on the UI

emit_new_file(status)

Emits the new_file_sig

Parameters

status (bool) – emits True if a new file has been asked by the user pressing the new file button on the UI

```
class pymodaq.utils.h5modules.saving.H5SaverBase(save_type='scan', backend='tables')
```

Object containing all methods in order to save data in a *hdf5 file* with a hierarchy compatible with the H5Browser. The saving parameters are contained within a **Parameter** object: self.settings that can be displayed on a UI using the widget self.settings_tree. At the creation of a new file, a node group named **Raw_datas** and represented by the attribute raw_group is created and set with a metadata attribute:

- 'type' given by the **save_type** class parameter

The root group of the file is then set with a few metadata:

- 'pymodaq_version' the current pymodaq version, e.g. 1.6.2
- 'file' the file name
- 'date' the current date
- 'time' the current time

All data will then be saved under this node in various groups

See also:

H5Browser

Parameters

- **h5_file** (pytables hdf5 file) – object used to save all data and metadata
- **h5_file_path** (str or Path) – Signal signal represented by a float. Is emitted each time the hardware reached the target position within the epsilon precision (see common_parameters variable)

- **save_type** (*str*) – an element of the enum module attribute SaveType * ‘scan’ is used for DAQScan module and should be used for similar application * ‘detector’ is used for DAQ_Viewer module and should be used for similar application * ‘custom’ should be used for customized applications

settings

Parameter instance (pyqtgraph) containing all settings (could be represented using the settings_tree widget)

Type

Parameter

settings_tree

Widget representing as a Tree structure, all the settings defined in the class preamble variable params

Type

ParameterTree

classmethod **find_part_in_path_and_subpath**(*base_dir*, *part*="", *create*=False, *increment*=True)

Find path from part time.

Parameters	Type	Description
<i>base_dir</i>	Path object	The directory to browse
<i>part</i>	string	The date of the directory to find/create
<i>create</i>	boolean	Indicate the creation flag of the directory

Returns

found path from part

Return type

Path object

get_last_scan()

Gets the last scan node within the h5_file and under the **raw_group**

Returns

scan_group

Return type

pytables group or None

get_scan_index()

return the scan group index in the “scan templating”: Scan000, Scan001 as an integer

init_file(*update_h5*=False, *custom_naming*=False, *addhoc_file_path*=None, *metadata*={})

Initializes a new h5 file. Could set the h5_file attributes as:

- a file with a name following a template if *custom_naming* is False and *addhoc_file_path* is None
- a file within a name set using a file dialog popup if *custom_naming* is True
- a file with a custom name if *addhoc_file_path* is a Path object or a path string

Parameters

- **update_h5** (*bool*) – create a new h5 file with name specified by other parameters if false try to open an existing file and will append new data to it
- **custom_naming** (*bool*) – if True, a selection file dialog opens to set a new file name

- **addhoc_file_path** (*Path* or *str*) – supplied name by the user for the new file
- **metadata** (*dict*) – dictionary with pair of key, value that should be saved as attributes of the root group

Returns

update_h5 – True if new file has been created, False otherwise

Return type

bool

load_file(*base_path=None, file_path=None*)

Opens a file dialog to select a h5file saved on disk to be used

Parameters

- **base_path** –
- **file_path** –

See also:

init_file()

classmethod set_current_scan_path(*base_dir, base_name='Scan', update_h5=False, next_scan_index=0, create_scan_folder=False, create_dataset_folder=True, curr_date=None, ind_dataset=None*)

Parameters

- **base_dir** –
- **base_name** –
- **update_h5** –
- **next_scan_index** –
- **create_scan_folder** –
- **create_dataset_folder** –

update_file_paths(*update_h5=False*)

Parameters

update_h5 (*bool*) – if True, will increment the file name and eventually the current scan index if False, get the current scan index in the h5 file

Returns

- **scan_path** (*Path*)
- **current_filename** (*str*)
- **dataset_path** (*Path*)

value_changed(*param*)

Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in *self._settings* is changed

Parameters

param (*Parameter*) – the parameter whose value just changed

Examples

```
>>> if param.name() == 'do_something':
>>>     if param.value():
>>>         print('Do something')
>>>         self.settings.child('main_settings', 'something_done').
→setValue(False)
```

They both inherits from the `ParameterManager MixIn` class that deals with `Parameter` and `ParameterTree`, see `saving_settings_fig`.

High Level saving/loading

Each PyMoDAQ's data type: `Axis`, `DataWithAxes`, `DataToExport` (see *What is PyMoDAQ's Data?*) is associated with its saver/loader counterpart. These objects ensures that all metadata necessary for an exact regeneration of the data is being saved at the correct location in the hdf5 file hierarchy. The `AxisSaverLoader`, `DataSaverLoader`, `DataToExportSaver` all derive from an abstract class: `DataManagement` allowing the manipulation of the nodes and making sure the data type is defined.

Base data class saver/loader

Created the 21/11/2022

@author: Sebastien Weber

class `pymodaq.utils.h5modules.data_saving.AxisSaverLoader(*args, **kwargs)`

Specialized Object to save and load Axis object to and from a h5file

Parameters

h5saver (`H5Saver`) –

data_type

The enum for this type of data, here 'axis'

Type

`DataType`

add_axis(where: `Union[Node, str]`, axis: `Axis`, `enlargeable=False`)

Write Axis info at a given position within a h5 file

Parameters

- **where** (`Union[Node, str]`) – the path of a given node or the node itself
- **axis** (`Axis`) – the Axis object to add as a node in the h5file
- **enlargeable** (`bool`) – Specify if the underlying array will be enlargebale

get_axes(where: `Union[Node, str]`) → `List[Axis]`

Return a list of Axis objects from the Axis Nodes hanging from (or among) a given Node

Parameters

where (`Union[Node, str]`) – the path of a given node or the node itself

Returns

`List[Axis]`

Return type

the list of all Axis object

load_axis(where: *Union*[Node, str]) → *Axis*

create an Axis object from the data and metadata at a given node if of data_type: 'axis'

Parameters

where (*Union*[Node, str]) – the path of a given node or the node itself

Return type

Axis

class pymodaq.utils.h5modules.data_saving.**DataManagement**(*args, **kwargs)

Base abstract class to be used for all specialized object saving and loading data to/from a h5file

data_type

The enum for this type of data, here abstract and should be redefined

Type

DataType

get_last_node_name(where: *Union*[str, Node]) → *Optional*[str]

Get the last node name among the ones already saved

Parameters

where (*Union*[Node, str]) – the path of a given node or the node itself

Returns

str

Return type

the name of the last saved node or None if none saved

class pymodaq.utils.h5modules.data_saving.**DataSaverLoader**(*args, **kwargs)

Specialized Object to save and load DataWithAxes object to and from a h5file

Parameters

h5saver (*H5Saver* or *Path* or str) –

data_type

The enum for this type of data, here 'data'

Type

DataType

add_data(where: *Union*[Node, str], data: *DataWithAxes*, save_axes=True, **kwargs)

Adds Array nodes to a given location adding eventually axes as others nodes and metadata

Parameters

- **where** (*Union*[Node, str]) – the path of a given node or the node itself
- **data** (*DataWithAxes*) –
- **save_axes** (*bool*) –

get_axes(where: *Union*[Node, str]) → *List*[*Axis*]

Parameters

where (*Union*[Node, str]) – the path of a given node or the node itself

get_data_arrays(*where*: *Union[Node, str]*, *with_bkg*=False, *load_all*=False) → *List[ndarray]*

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **with_bkg** (*bool*) – If True try to load background node and return the array with background subtraction
- **load_all** (*bool*) – If True load all similar nodes hanging from a parent

Return type

list of *ndarray*

isopen() → *bool*

Get the opened status of the underlying hdf5 file

load_data(*where*, *with_bkg*=False, *load_all*=False) → *DataWithAxes*

Return a *DataWithAxes* object from the Data and Axis Nodes hanging from (or among) a given Node

Does not include navigation axes stored elsewhere in the h5file. The node path is stored in the *DatWithAxis* using the attribute path

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **with_bkg** (*bool*) – If True try to load background node and return the data with background subtraction
- **load_all** (*bool*) – If True, will load all data hanging from the same parent node

See also:

load_data

class *pymodaq.utils.h5modules.data_saving.DataToExportSaver*(*h5saver*: *Union[H5Saver, Path, str]*)

Object used to save *DataToExport* object into a h5file following the PyMoDAQ convention

Parameters

h5saver (*H5Saver*) –

add_data(*where*: *Union[Node, str]*, *data*: *DataToExport*, *settings_as_xml*="", *metadata*=None, ***kwargs*)

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **data** (*DataToExport*) –
- **settings_as_xml** (*str*) – The settings parameter as an XML string
- **metadata** (*dict*) – all extra metadata to be saved in the group node where data will be saved

static channel_formatter(*ind*: *int*)

All *DataWithAxes* included in the *DataToExport* will be saved into a channel group indexed and formatted as below

isopen() → *bool*

Get the opened status of the underlying hdf5 file

Specific data class saver/loader

Some more dedicated objects are derived from the objects above. They allow to add background data, Extended arrays (arrays that will be populated after creation, for instance for a scan) and Enlargeable arrays (whose final length is not known at the moment of creation, for instance when logging or continuously saving)

Created the 21/11/2022

@author: Sebastien Weber

```
class pymodaq.utils.h5modules.data_saving.BkgSaver(*args, **kwargs)
```

Specialized Object to save and load DataWithAxes background object to and from a h5file

Parameters

hsaver ([H5Saver](#)) –

data_type

The enum for this type of data, here ‘bkg’

Type

DataType

```
class pymodaq.utils.h5modules.data_saving.DataEnlargeableSaver(*args, **kwargs)
```

Specialized Object to save and load enlargeable DataWithAxes saved object to and from a h5file

Particular case of DataND with a single *nav_indexes* parameter will be appended as chunks of signal data

Parameters

hsaver ([H5Saver](#)) –

data_type

The enum for this type of data, here ‘data_enlargeable’

Type

DataType

Notes

To be used to save data from a timed logger (DAQViewer continuous saving or DAQLogger extension) or from an adaptive scan where the final shape is unknown or other module that need this feature

add_data (where: *Union[Node, str]*, data: *DataWithAxes*, axis_values: *Optional[Iterable[float]] = None*)

Append data to an enlargeable array node

Data of dim (0, 1 or 2) will be just appended to the enlargeable array.

Uniform DataND with one navigation axis of length (Lnav) will be considered as a collection of Lnav signal data of dim (0, 1 or 2) and will therefore be appended as Lnav signal data

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **data** (*DataWithAxes*) –
- **axis_values** (*optional, list of floats*) – the new spread axis values added to the data if None the axes are not added to the h5 file

```
class pymodaq.utils.h5modules.data_saving.DataExtendedSaver(*args, **kwargs)
```

Specialized Object to save and load DataWithAxes saved object to and from a h5file in extended arrays

Parameters

- **h5saver** ([H5Saver](#)) –
- **extended_shape** ([Tuple](#)[[int](#)]) – the extra shape compared to the data the h5array will have

data_type

The enum for this type of data, here 'data'

Type

[DataType](#)

```
add_data(where: Union[Node, str], data: DataWithAxes, indexes: List[int],  
         distribution=DataDistribution.uniform)
```

Adds given [DataWithAxes](#) at a location within the initialized h5 array

Parameters

- **where** ([Union](#)[[Node](#), [str](#)]) – the path of a given node or the node itself
- **data** ([DataWithAxes](#)) –
- **indexes** ([Iterable](#)[[int](#)]) – indexes where to save data in the init h5array (should have the same length as `extended_shape` and with values coherent with this shape)

```
class pymodaq.utils.h5modules.data_saving.DataToExportEnlargeableSaver(h5saver: H5Saver,  
                               enl_axis_names: Optional[Iterable[str]]  
                               = None, enl_axis_units: Optional[Iterable[str]]  
                               = None, axis_name: str  
                               = 'nav axis', axis_units: str = '')
```

Generic object to save [DataToExport](#) objects in an enlargeable h5 array

The next enlarged value should be specified in the `add_data` method

Parameters

- **h5saver** ([H5Saver](#)) –
- **enl_axis_names** ([Iterable](#)[[str](#)]) – The names of the enlargeable axis, default ['nav_axis']
- **enl_axis_units** ([Iterable](#)[[str](#)]) – The names of the enlargeable axis, default ['']
- **axis_name** ([str](#), deprecated use `enl_axis_names`) – the name of the enlarged axis array
- **axis_units** ([str](#), deprecated use `enl_axis_units`) – the units of the enlarged axis array

```
add_data(where: Union[Node, str], data: DataToExport, axis_values: Optional[List[Union[float, ndarray]]]  
         = None, axis_value: Optional[Union[float, ndarray]] = None, settings_as_xml="",  
         metadata=None)
```

Parameters

- **where** ([Union](#)[[Node](#), [str](#)]) – the path of a given node or the node itself

- **data** (`DataToExport`) – The data to be saved into an enlargeable array
- **axis_values** (*iterable float or np.ndarray*) – The next value (or values) of the enlarged axis
- **axis_value** (*float or np.ndarray #deprecated in 4.2.0, use axis_values*) – The next value (or values) of the enlarged axis
- **settings_as_xml** (*str*) – The settings parameter as an XML string
- **metadata** (*dict*) – all extra metadata to be saved in the group node where data will be saved

```
class pymodaq.utils.h5modules.data_saving.DataToExportExtendedSaver(h5saver: H5Saver,
                                                                    extended_shape:
                                                                    Tuple[int])
```

Object to save DataToExport at given indexes within arrays including extended shape

Mostly used for data generated from the DAQScan

Parameters

- **h5saver** (`H5Saver`) –
- **extended_shape** (*Tuple[int]*) – the extra shape compared to the data the h5array will have

```
add_data(where: Union[Node, str], data: DataToExport, indexes: Iterable[int],
         distribution=DataDistribution.uniform, settings_as_xml="", metadata={})
```

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **data** (`DataToExport`) –
- **indexes** (*List[int]*) – indexes where to save data in the init h5array (should have the same length as extended_shape and with values coherent with this shape)
- **settings_as_xml** (*str*) – The settings parameter as an XML string
- **metadata** (*dict*) – all extra metadata to be saved in the group node where data will be saved

```
add_nav_axes(where: Union[Node, str], axes: List[Axis])
```

Used to add navigation axes related to the extended array

Notes

For instance the scan axes in the DAQScan

```
class pymodaq.utils.h5modules.data_saving.DataToExportTimedSaver(h5saver: H5Saver)
```

Specialized DataToExportEnlargeableSaver to save data as a function of a time axis

Only one element can be added at a time, the time axis value are enlarged using the data to be added timestamp

Notes

This object is made for continuous saving mode of DAQViewer and logging to h5file for DAQLogger

add_data(where: *Union[Node, str]*, data: *DataToExport*, settings_as_xml="", metadata=None, **kwargs)

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **data** (*DataToExport*) – The data to be saved into an enlargeable array
- **axis_values** (*iterable float or np.ndarray*) – The next value (or values) of the enlarged axis
- **axis_value** (*float or np.ndarray #deprecated in 4.2.0, use axis_values*) – The next value (or values) of the enlarged axis
- **settings_as_xml** (*str*) – The settings parameter as an XML string
- **metadata** (*dict*) – all extra metadata to be saved in the group node where data will be saved

Specialized loading

Data saved from a DAQ_Scan will naturally include navigation axes shared between many different DataWithAxes (as many as detectors/channels/ROIs). They are therefore saved at the root of the scan node and cannot be retrieved using the standard data loader. Hence this DataLoader object.

class pymodaq.utils.h5modules.data_saving.**DataLoader**(h5saver: *Union[H5Saver, Path]*)

Specialized Object to load DataWithAxes object from a h5file

On the contrary to DataSaverLoader, does include navigation axes stored elsewhere in the h5file (for instance if saved from the DAQ_Scan)

Parameters

h5saver (*H5Saver*) –

Attributes

h5saver

Methods

get_nav_group(where)

param where

the path of a given node or the node itself

get_node(where[, name])

Convenience method to get node

load_data(where[, with_bkg, load_all])

Load data from a node (or channel node)

walk_nodes([where])

Return a Node generator iterating over the h5file content

close_file	
load_all	

get_nav_group(where: *Union*[Node, str]) → *Optional*[Node]

Parameters

where (*Union*[Node, str]) – the path of a given node or the node itself

Returns

- **GROUP** (returns the group named *SPECIAL_GROUP_NAMES*['nav_axes'] holding all NavAxis for)
- those data

See also:

SPECIAL_GROUP_NAMES

get_node(where: *Union*[Node, str], name: *Optional*[str] = None) → Node

Convenience method to get node

load_data(where: *Union*[Node, str], with_bkg=False, load_all=False) → DataWithAxes

Load data from a node (or channel node)

Loaded data contains also nav_axes if any and with optional background subtraction

Parameters

- **where** (*Union*[Node, str]) – the path of a given node or the node itself
- **with_bkg** (*bool*) – If True will attempt to subtract a background data node before loading
- **load_all** (*bool*) – If True, will load all data hanging from the same parent node

walk_nodes(where: *Union*[str, Node] = '/')

Return a Node generator iterating over the h5file content

Browsing Data

Using the *H5Backend* it is possible to write scripts to easily access a hdf5 file content. However, PyMoDAQ includes a dedicated hdf5 viewer understanding dedicated metadata and therefore displaying nicely the content of the file, see *H5Browser*. Two objects can be used to browse data: *H5BrowserUtil* and *H5Browser*. *H5BrowserUtil* gives you methods to quickly (in a script) get info and data from your file while the *H5Browser* adds a UI to interact with the hdf5 file.

Created the 15/11/2022

@author: Sebastien Weber

class pymodaq.utils.h5modules.browsing.**H5Browser**(parent: *QMainWindow*, h5file=None, h5file_path=None, backend='tables')

UI used to explore h5 files, plot and export subdatas

Parameters

- **parent** (*QtWidgets container*) – either a QWidget or a QMainWindow
- **h5file** (*h5file instance*) – exact type depends on the backend
- **h5file_path** (*str or Path*) – if specified load the corresponding file, otherwise open a select file dialog
- **backend** (*str*) – either 'tables', 'h5py' or 'h5pyd'

See also:

H5Backend, H5Backend

add_comments(*status: bool, comment=""*)

Add comments to a node

Parameters

- **status** (*bool*) –
- **comment** (*str*) – The comment to be added in a comment attribute to the current node path

See also:

current_node_path

check_version()

Check version of PyMoDAQ to assert if file is compatible or not with the current version of the Browser

export_data()

Opens a dialog to export data

See also:

H5BrowserUtil.export_data

get_tree_node_path()

Get the node path of the currently selected node in the UI

populate_tree()

Init the ui-tree and store data into calling the h5_tree_to_Qtree convertor method

See also:

h5tree_to_QTree, update_status

quit_fun()

setup_actions()

Method where to create actions to be subclassed. Mandatory

Examples

```
>>> self.add_action('Quit', 'close2', "Quit program")
>>> self.add_action('Grab', 'camera', "Grab from camera", checkable=True)
>>> self.add_action('Load', 'Open', "Load target file (.h5, .png, .jpg) or
↳data from camera", checkable=False)
>>> self.add_action('Save', 'SaveAs', "Save current data", checkable=False)
```

See also:

ActionManager.add_action

show_h5_data(*item, with_bkg=False, plot_all=False*)

Parameters

- **item** –

- **with_bkg** –
- **plot_all** –

class pymodaq.utils.h5modules.browsing.**H5BrowserUtil**(*backend='tables'*)

Utility object to interact and get info and data from a hdf5 file

Inherits H5Backend and all its functionalities

Parameters

backend (*str*) – The used hdf5 backend: either tables, h5py or h5pyd

export_data(*node_path=''*, *filesavename: str = 'datafile.h5'*, *filter=None*)

Initialize the correct exporter and export the node

get_h5_attributes(*node_path*)

get_h5file_scans(*where=''*)

Get the list of the scan nodes in the file

Parameters

where (*str*) – the path in the file

Returns

dict with keys: scan_name, path (within the file) and data (the live scan png image)

Return type

list of dict

Module savers

Created the 23/11/2022

@author: Sebastien Weber

class pymodaq.utils.h5modules.module_saving.**ActuatorSaver**(*args, **kwargs)

Implementation of the ModuleSaver class dedicated to DAQ_Move modules

Parameters

- **h5saver** –
- **module** –

class pymodaq.utils.h5modules.module_saving.**DetectorEnlargeableSaver**(*args, **kwargs)

Implementation of the ModuleSaver class dedicated to DAQ_Viewer modules in order to save enlargeable data

Parameters

module –

class pymodaq.utils.h5modules.module_saving.**DetectorExtendedSaver**(*args, **kwargs)

Implementation of the ModuleSaver class dedicated to DAQ_Viewer modules in order to save enlargeable data

Parameters

module –

class pymodaq.utils.h5modules.module_saving.**DetectorSaver**(*args, **kwargs)

Implementation of the ModuleSaver class dedicated to DAQ_Viewer modules

Parameters

module –

add_bkg(*where*: *Union*[*Node*, *str*], *data_bkg*: *DataToExport*)

Adds a *DataToExport* as a background node in the h5file

Parameters

- **where** (*Union*[*Node*, *str*]) – the path of a given node or the node itself
- **data_bkg** (*DataToExport*) – The data to be saved as background

class pymodaq.utils.h5modules.module_saving.**LoggerSaver**(*args, **kwargs)

Implementation of the *ModuleSaver* class dedicated to *H5Logger* module

H5Logger is the special logger to h5file of the *DAQ_Logger* extension

Parameters

- **h5saver** –
- **module** –

add_data(*dte*: *DataToExport*)

Add data to it's corresponding control module

The name of the control module is the *DataToExport* name attribute

class pymodaq.utils.h5modules.module_saving.**ModuleSaver**(*args, **kwargs)

Abstract base class to save info and data from main modules (*DAQScan*, *DAQViewer*, *DAQMove*, ...)

flush()

Flush the underlying file

get_last_node(*where*: *Optional*[*Union*[*Node*, *str*]] = *None*)

Get the last node corresponding to this particular *Module* instance

Parameters

- **where** (*Union*[*Node*, *str*]) – the path of a given node or the node itself
- **new** (*bool*) – if *True* force the creation of a new indexed node of this class type if *False* return the last node (or create one if *None*)

Returns

GROUP

Return type

the *Node* associated with this module which should be a *GROUP* node

get_set_node(*where*: *Optional*[*Union*[*Node*, *str*]] = *None*, *name*: *Optional*[*str*] = *None*) → *GROUP*

Get or create the node corresponding to this particular *Module* instance

Parameters

- **where** (*Union*[*Node*, *str*]) – the path of a given node or the node itself
- **new** (*bool*) – if *True* force the creation of a new indexed node of this class type if *False* return the last node (or create one if *None*)

Returns

GROUP

Return type

the *Node* associated with this module which should be a *GROUP* node

```
class pymodaq.utils.h5modules.module_saving.ScanSaver(*args, **kwargs)
```

Implementation of the ModuleSaver class dedicated to DAQScan module

Parameters

- **h5saver** –
- **module** –

```
get_set_node(where: Optional[Union[Node, str]] = None, new=False) → GROUP
```

Get the last group scan node

Get the last Scan Group or create one get the last Scan Group if: * there is one already created * new is False

Parameters

- **where** (*Union[Node, str]*) – the path of a given node or the node itself
- **new** (*bool*) –

Returns

GROUP

Return type

the GROUP associated with this module

Scanner module and classes

Summary of the classes in the scanner module

<code>Scanner</code> ([parent_widget, scanner_items, ...])	Main Object to define a PyMoDAQ scan and create a UI to set it
--	--

The scanner module contains all functionalities to defines a particular scan see scanner_paragraph.

```
class pymodaq.utils.scanner.Scanner(parent_widget: QtWidgets.QWidget = None, scanner_items={},  
                                   actuators: List[DAQ_Move] = [])
```

Main Object to define a PyMoDAQ scan and create a UI to set it

Parameters

- **parent_widget** (*QtWidgets.QWidget*) –
- **scanner_items** (*list of GraphicItems*) – used by ScanSelector for chosing scan area or linear traces
- **actuators** (*List[DAQ_Move]*) – list actuators names

See also:

ScanSelector, ScannerBase, TableModelSequential, TableModelTabular, pymodaq_types.
TableViewCustom

Attributes

`actuators`

list of str: Returns as a list the name of the selected actuators to describe the actual scan

axes_indexes
axes_unique
distribution
n_axes
n_steps
positions
scan_sub_type
scan_type
scanner

Methods

<code>get_indexes_from_scan_index(scan_index)</code>	To be reimplemented.
<code>get_scan_info()</code>	Get a summary of the configured scan as a ScanInfo object
<code>get_scanner_sub_settings()</code>	Get the current ScannerBase implementation's settings
<code>positions_at(index)</code>	Extract the actuators positions at a given index in the scan as a DataToExport of DataActuators
<code>set_scan()</code>	Process the settings options to calculate the scan positions
<code>set_scan_type_and_subtypes(scan_type, ...)</code>	Convenience function to set the main scan type
<code>value_changed(param)</code>	Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

connect_things	
get_nav_axes	
get_scan_shape	
save_scanner_settings	
scanner_updated_signal	
set_scan_from_settings	
set_scanner	
setup_ui	
update_from_scan_selector	

get_indexes_from_scan_index(*scan_index: int*) → [Tuple\[int\]](#)

To be reimplemented. Calculations of indexes within the scan

get_scan_info() → [ScanInfo](#)

Get a summary of the configured scan as a ScanInfo object

get_scanner_sub_settings()

Get the current ScannerBase implementation's settings

positions_at(*index: int*) → [DataToExport](#)

Extract the actuators positions at a given index in the scan as a DataToExport of DataActuators

set_scan()

Process the settings options to calculate the scan positions

Returns**bool****Return type**True if the processed number of steps is **higher** than the configured number of steps**set_scan_type_and_subtypes**(*scan_type*: *str*, *scan_subtype*: *str*)

Convenience function to set the main scan type

Parameters

- **scan_type** (*str*) – one of registered Scanner main identifier
- **scan_subtype** (*list of str or None*) – one of registered Scanner second identifier for a given main identifier

See also:

ScannerFactory

value_changed(*param*: *Parameter*)

Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

Parameters**param** (*Parameter*) – the parameter whose value just changed**Examples**

```
>>> if param.name() == 'do_something':
>>>     if param.value():
>>>         print('Do something')
>>>         self.settings.child('main_settings', 'something_done').
→setValue(False)
```

property actuators

Returns as a list the name of the selected actuators to describe the actual scan

Typelist of *str***Managers**

API of the various managers, special classes to deals with QAction, Paramaters, ControlModules...

<i>addaction</i> ([name, icon_name, tip, checkable, ...])	Create a new action and add it eventually to a toolbar and a menu
<i>QAction</i> (*args, **kwargs)	QAction subclass to mimic signals as pushbuttons.
<i>ActionManager</i> ([toolbar, menu])	Mixin Class to be used by all UserInterface to manage their QActions and the action they are connected to
<i>ParameterManager</i> ([settings_name, action_list])	Class dealing with Parameter and ParameterTree
<i>ModulesManager</i> ([detectors, actuators, ...])	Class to manage DAQ_Viewers and DAQ_Moves with UI to select some

```
class pymodaq.utils.managers.action_manager.QAction(*args, **kwargs)
```

QAction subclass to mimic signals as pushbuttons. Done to be sure of backcompatibility when I moved from pushbuttons to QAction

Attributes

clicked

Methods

click	
connect_to	
set_icon	

```
pymodaq.utils.managers.action_manager.addaction(name: str = "", icon_name: str = "", tip="",
                                                checkable=False, checked=False, slot:
                                                Optional[Callable] = None, toolbar:
                                                Optional[QToolBar] = None, menu:
                                                Optional[QMenu] = None, visible=True,
                                                shortcut=None, enabled=True)
```

Create a new action and add it eventually to a toolbar and a menu

Parameters

- **name** (*str*) – Displayed name if should be displayed (for instance in menus)
- **icon_name** (*str*) – png file name to produce the icon
- **tip** (*str*) – a tooltip to be displayed when hovering above the action
- **checkable** (*bool*) – set the checkable state of the action
- **checked** (*bool*) – set the current state of the action
- **slot** (*callable*) – Method or function that will be called when the action is triggered
- **toolbar** (*QToolBar*) – a toolbar where action should be added.
- **menu** (*QMenu*) – a menu where action should be added.
- **visible** (*bool*) – display or not the action in the toolbar/menu
- **shortcut** (*str*) – a string defining a shortcut for this action
- **enabled** (*bool*) – set the enabled state

```
class pymodaq.utils.managers.action_manager.ActionManager(toolbar=None, menu=None)
```

MixIn Class to be used by all UserInterface to manage their QActions and the action they are connected to

Parameters

- **toolbar** (*QToolBar*, *optional*) – The toolbar to use as default
- **menu** (*QMenu*, *option*) – The menu to use as default

Attributes

actions

menu

Get the default menu

toolbar

Get the default toolbar

Methods

<code>add_action([short_name, name, icon_name, ...])</code>	Create a new action and add it to toolbar and menu
<code>add_widget(short_name, klass, *args[, tip, ...])</code>	Create and add a widget to a toolbar
<code>affect_to(action_name, obj)</code>	Affect action to an object either a toolbar or a menu
<code>connect_action(name, slot[, connect, ...])</code>	Connect (or disconnect) the action referenced by name to the given slot
<code>get_action(name)</code>	Getter of a given action
<code>has_action(action_name)</code>	Check if an action has been defined :param action_name: The action name as defined in setup_actions :type action_name: str
<code>is_action_checked</code>	Dispatch methods based on type signature
<code>is_action_enabled</code>	Dispatch methods based on type signature
<code>is_action_visible</code>	Dispatch methods based on type signature
<code>set_action_checked</code>	Dispatch methods based on type signature
<code>set_action_enabled</code>	Dispatch methods based on type signature
<code>set_action_text(action_name, text)</code>	Convenience method to set the displayed text on an action
<code>set_action_visible</code>	Dispatch methods based on type signature
<code>set_menu(menu)</code>	affect a menu to self
<code>set_toolbar(toolbar)</code>	affect a toolbar to self
<code>setup_actions()</code>	Method where to create actions to be subclassed.

add_action(short_name: *str* = "", name: *str* = "", icon_name: *str* = "", tip="", checkable=False, checked=False, toolbar=None, menu=None, visible=True, shortcut=None, auto_toolbar=True, auto_menu=True, enabled=True)

Create a new action and add it to toolbar and menu

Parameters

- **short_name** (*str*) – the name as referenced in the dict self.actions
- **name** (*str*) – Displayed name if should be displayed in
- **icon_name** (*str*) – png file name to produce the icon
- **tip** (*str*) – a tooltip to be displayed when hovering above the action
- **checkable** (*bool*) – set the checkable state of the action
- **checked** (*bool*) – set the current state of the action
- **toolbar** (*QToolBar*) – a toolbar where action should be added. Actions can also be added later see *affect_to*
- **menu** (*QMenu*) – a menu where action should be added. Actions can also be added later see *affect_to*
- **visible** (*bool*) – display or not the action in the toolbar/menu
- **auto_toolbar** (*bool*) – if True add this action to the defined toolbar
- **auto_menu** (*bool*) – if True add this action to the defined menu
- **enabled** (*bool*) – set the enabled state of this action

See also:

[affect_to](#), `pymodaq.resources.QtDesigner_Ressources.Icon_Library`, `pymodaq.utils.managers.action_manager.add_action`

add_widget(*short_name*, *klass*: *Union*[*str*, *QWidget*], **args*, *tip*="", *toolbar*: *Optional*[*QToolBar*] = *None*, *visible*=*True*, *signal_str*=*None*, *slot*: *Optional*[*Callable*] = *None*, ***kwargs*)

Create and add a widget to a toolbar

Parameters

- **short_name** (*str*) – the name as referenced in the dict `self.actions`
- **klass** (*str* or *QWidget*) – should be a custom widget class or the name of a standard widget of *QWidgets*
- **args** (*list*) – variable arguments passed as is to the widget constructor
- **tip** (*str*) – a tooltip to be displayed when hovering above the widget
- **toolbar** (*QToolBar*) – a toolbar where the widget should be added.
- **visible** (*bool*) – display or not the action in the toolbar/menu
- **signal_str** (*str*) – an attribute of type *Signal* of the widget
- **slot** (*Callable*) – a callable connected to the signal
- **kwargs** (*dict*) – variable named arguments passed as is to the widget constructor

Return type

`QtWidgets.QWidget`

affect_to(*action_name*, *obj*: *Union*[*QToolBar*, *QMenu*])

Affect action to an object either a toolbar or a menu

Parameters

- **action_name** (*str*) – The action name as defined in `setup_actions`
- **obj** (*QToolBar* or *QMenu*) – The object where to add the action

connect_action(*name*, *slot*, *connect*=*True*, *signal_name*="")

Connect (or disconnect) the action referenced by name to the given slot

Parameters

- **name** (*str*) – key of the action as referenced in the `self._actions` dict
- **slot** (*method*) – a method/function
- **connect** (*bool*) – if *True* connect the trigger signal of the action to the defined slot else disconnect it
- **signal_name** (*str*) – try to use it as a signal (for widgets added...) otherwise use the *triggered* signal

get_action(*name*) → *QAction*

Getter of a given action

Parameters

- **name** (*str*) – The action name as defined in `setup_actions`

Return type

QAction

has_action(*action_name*) → bool

Check if an action has been defined :param action_name: The action name as defined in setup_actions
:type action_name: str

Returns

bool

Return type

True if the action exists, False otherwise

set_action_text(*action_name: str, text: str*)

Convenience method to set the displayed text on an action

Parameters

- **action_name** (*str*) – The action name as defined in setup_actions
- **text** (*str*) – The text to display

set_menu(*menu*)

affect a menu to self

Parameters

menu – QtWidgets.QMenu

set_toolbar(*toolbar*)

affect a toolbar to self

Parameters

toolbar – QtWidgets.QToolBar

setup_actions()

Method where to create actions to be subclassed. Mandatory

Examples

```
>>> self.add_action('Quit', 'close2', "Quit program")
>>> self.add_action('Grab', 'camera', "Grab from camera", checkable=True)
>>> self.add_action('Load', 'Open', "Load target file (.h5, .png, .jpg) or ↵
↳data from camera", checkable=False)
>>> self.add_action('Save', 'SaveAs', "Save current data", checkable=False)
```

See also:

[*ActionManager.add_action*](#)

property menu

Get the default menu

property toolbar

Get the default toolbar

```
class pymodaq.utils.managers.parameter_manager.ParameterManager(settings_name: Optional[str] =
None, action_list: tuple = ('save',
'update', 'load'))
```

Class dealing with Parameter and ParameterTree

params

Defining the Parameter tree like structure

Type

list of dicts

settings_name

The particular name to give to the object parent Parameter (self.settings)

Type

str

settings

The higher level (parent) Parameter

Type

Parameter

settings_tree

widget Holding a ParameterTree and a toolbar for interacting with the tree

Type

QWidget

tree

the underlying ParameterTree

Type

ParameterTree

Attributes

settings

settings_tree

tree

Methods

<i>child_added</i> (param, data)	Non-mandatory method to be subclassed for actions to perform when a param has been added in self.settings
<i>load_settings_slot</i> ([file_path])	Method to load settings into the parameter using a xml file extension.
<i>param_deleted</i> (param)	Non-mandatory method to be subclassed for actions to perform when one of the param in self.settings has been deleted
<i>save_settings_slot</i> ([file_path])	Method to save the current settings using a xml file extension.
<i>update_settings_slot</i> ([file_path])	Method to update settings using a xml file extension.
<i>value_changed</i> (param)	Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

create_parameter	
parameter_tree_changed	

child_added(*param*, *data*)

Non-mandatory method to be subclassed for actions to perform when a param has been added in `self.settings`

Parameters

- **param** (*Parameter*) – the parameter where child will be added
- **data** (*Parameter*) – the child parameter

load_settings_slot(*file_path*: *Optional[Path]* = *None*)

Method to load settings into the parameter using a xml file extension.

The starting directory is the user config folder with a subfolder called settings folder

Parameters

- **file_path** (*Path*) – Path like object pointing to a xml file encoding a *Parameter* object
If *None*, opens a file explorer window to pick manually a file

param_deleted(*param*)

Non-mandatory method to be subclassed for actions to perform when one of the param in `self.settings` has been deleted

Parameters

- **param** (*Parameter*) – the parameter that has been deleted

save_settings_slot(*file_path*: *Optional[Path]* = *None*)

Method to save the current settings using a xml file extension.

The starting directory is the user config folder with a subfolder called settings folder

Parameters

- **file_path** (*Path*) – Path like object pointing to a xml file encoding a *Parameter* object
If *None*, opens a file explorer window to save manually a file

update_settings_slot(*file_path*: *Optional[Path]* = *None*)

Method to update settings using a xml file extension.

The file should define the same settings structure (names and children)

The starting directory is the user config folder with a subfolder called settings folder

Parameters

- **file_path** (*Path*) – Path like object pointing to a xml file encoding a *Parameter* object
If *None*, opens a file explorer window to pick manually a file

value_changed(*param*)

Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in `self._settings` is changed

Parameters

- **param** (*Parameter*) – the parameter whose value just changed

Examples

```
>>> if param.name() == 'do_something':
>>>     if param.value():
>>>         print('Do something')
>>>         self.settings.child('main_settings', 'something_done').
↪setValue(False)
```

```
class pymodaq.utils.managers.modules_manager.ModulesManager(detectors=[], actuators=[],
                                                             selected_detectors=[],
                                                             selected_actuators=[], **kwargs)
```

Class to manage DAQ_Viewers and DAQ_Moves with UI to select some

Easier to connect control modules signals to slots, test, ...

Parameters

- **detectors** (*list of DAQ_Viewer*) –
- **actuators** (*list of DAQ_Move*) –
- **selected_detectors** (*list of DAQ_Viewer*) – sublist of detectors
- **selected_actuators** (*list of DAQ_Move*) – sublist of actuators

Attributes

Nactuators

Get the number of selected actuators

Ndetectors

Get the number of selected detectors

actuators

Get the list of selected actuators

actuators_all

Get the list of all actuators

actuators_name

Get all the names of the actuators

detectors

Get the list of selected detectors

detectors_all

Get the list of all detectors

detectors_name

Get all the names of the detectors

modules

Get the list of all detectors and actuators

modules_all

Get the list of all detectors and actuators

selected_actuators_name

Get/Set the names of the selected actuators

selected_detectors_name

Get/Set the names of the selected detectors

Methods

<code>connect_actuators([connect, slot, signal])</code>	Connect the selected actuators signal to a given or default slot
<code>connect_detectors([connect, slot])</code>	Connect selected DAQ_Viewers's grab_done_signal to the given slot
<code>get_det_data_list()</code>	Do a snap of selected detectors, to get the list of all the data and processed data
<code>get_mod_from_name(name[, mod])</code>	Getter of a given module from its name (title)
<code>get_mods_from_names(names[, mod])</code>	Getter of a list of given modules from their name (title)
<code>get_names(modules)</code>	Get the titles of a list of Control Modules
<code>get_selected_probed_data([dim])</code>	Get the name of selected data names of a given dimensionality
<code>grab_datas(**kwargs)</code>	Do a single grab of connected and selected detectors
<code>move_actuators(dte_act[, mode, polling])</code>	will apply positions to each currently selected actuators.
<code>order_positions(positions)</code>	Reorder the content of the DataToExport given the order of the selected actuators
<code>set_actuators(actuators, selected_actuators)</code>	Populates actuators and the subset to be selected in the UI
<code>set_detectors(detectors, selected_detectors)</code>	Populates detectors and the subset to be selected in the UI
<code>test_move_actuators()</code>	Do a move of selected actuator
<code>value_changed(param)</code>	Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

actuators_changed	
det_done	
det_done_signal	
detectors_changed	
move_done	
move_done_signal	
reset_signals	
show_only_control_modules	
timeout_signal	

connect_actuators(*connect=True, slot=None, signal='move_done'*)

Connect the selected actuators signal to a given or default slot

Parameters

- **connect** (*bool*) –
- **slot** (*builtin_function_or_method*) – method or function the chosen signal will be connected to if None, then the default move_done slot is used
- **signal** (*str*) – What kind of signal is to be used:
 - 'move_done' will connect the *move_done_signal* to the slot
 - 'current_value' will connect the 'current_value_signal' to the slot

See also:

`move_done()`

connect_detectors(*connect=True, slot=None*)

Connect selected DAQ_Viewers's grab_done_signal to the given slot

Parameters

- **connect** (*bool*) – if True, connect to the given slot (or default slot) if False, disconnect all detectors (not only the currently selected ones. This is made because when selected detectors changed if you only disconnect those one, the previously connected ones will stay connected)
- **slot** (*method*) – A method that should be connected, if None self.det_done is connected by default

get_det_data_list()

Do a snap of selected detectors, to get the list of all the data and processed data

get_mod_from_name(*name, mod='det'*) → Union[DAQ_Move, DAQ_Viewer]

Getter of a given module from its name (title)

Parameters

- **name** (*str*) –
- **mod** (*str*) – either 'det' for DAQ_Viewer modules or 'act' for DAQ_Move modules

get_mods_from_names(*names, mod='det'*)

Getter of a list of given modules from their name (title)

Parameters

- **names** (*list of str*) –
- **mod** (*str*) – either 'det' for DAQ_Viewer modules or 'act' for DAQ_Move modules

classmethod get_names(*modules*)

Get the titles of a list of Control Modules

Parameters

modules (*list of DAQ_Move and/or DAQ_Viewer*) –

get_selected_probed_data(*dim='0D'*)

Get the name of selected data names of a given dimensionality

Parameters

dim (*str*) – either '0D', '1D', '2D' or 'ND'

grab_datas(***kwargs*)

Do a single grab of connected and selected detectors

move_actuators(*dte_act: DataToExport, mode='abs', polling=True*) → DataToExport

will apply positions to each currently selected actuators. By Default the mode is absolute but can be

Parameters

- **dte_act** (*DataToExport*) – the DataToExport of position to apply. Its length must be equal to the number of selected actuators
- **mode** (*str*) – either 'abs' for absolute positioning or 'rel' for relative

- **polling** (*bool*) – if True will wait for the selected actuators to reach their target positions (they have to be connected to a method checking for the position and letting the program know the move is done (default connection is this object *move_done* method))

Return type

DataToExport with the selected actuators's name as key and current actuators's value as value

order_positions(*positions*: DataToExport)

Reorder the content of the DataToExport given the order of the selected actuators

set_actuators(*actuators*, *selected_actuators*)

Populates actuators and the subset to be selected in the UI

set_detectors(*detectors*, *selected_detectors*)

Populates detectors and the subset to be selected in the UI

test_move_actuators()

Do a move of selected actuator

value_changed(*param*)

Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

Parameters

param (*Parameter*) – the parameter whose value just changed

Examples

```
>>> if param.name() == 'do_something':
>>>     if param.value():
>>>         print('Do something')
>>>         self.settings.child('main_settings', 'something_done').
→setValue(False)
```

property Nactuators

Get the number of selected actuators

property Ndetectors

Get the number of selected detectors

property actuators: List[DAQ_Move]

Get the list of selected actuators

property actuators_all

Get the list of all actuators

property actuators_name

Get all the names of the actuators

property detectors: List[DAQ_Viewer]

Get the list of selected detectors

property detectors_all

Get the list of all detectors

property detectors_name

Get all the names of the detectors

property modules

Get the list of all detectors and actuators

property modules_all

Get the list of all detectors and actuators

property selected_actuators_name: List[str]

Get/Set the names of the selected actuators

property selected_detectors_name

Get/Set the names of the selected detectors

Data Viewers

The data viewers are classes devoted to data display from scalar data up to 4 dimensions data. All the viewers inherits from the base class *ViewerBase* and then offers options and interactions depending their dimensionality

Summary of the data viewers classes

<code>pymodaq.utils.plotting.data_viewers.viewer0D.Viewer0D(...)</code>	this plots 0D data on a plotwidget with history.
<code>pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D(...)</code>	DataWithAxis of type Data1D can be plotted using this data viewer
<code>pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D(...)</code>	Object managing plotting and manipulation of 2D data using a View2D
<code>pymodaq.utils.plotting.data_viewers.viewerND.ViewerND(...)</code>	

Methods

```
class pymodaq.utils.plotting.data_viewers.viewer0D.Viewer0D(parent=None, title="",
                                                             show_toolbar=True,
                                                             no_margins=False)
```

this plots 0D data on a plotwidget with history. Display as numbers in a table is possible.

Datas and measurements are then exported with the signal `data_to_export_signal`

Attributes

labels

Methods

<code>update_colors</code>

```
class pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D(parent: Optional[QWidget] = None,
                                                             title="", show_toolbar=True,
                                                             no_margins=False, flip_axes=False)
```

DataWithAxis of type Data1D can be plotted using this data viewer

show_data:

parameter: * dwa: a DataWithaxis * scatter_dwa: an optional extra DataWithAxis to be plotted with scatter points

it could define extra_attributes such as symbol: str (to define the symbol layout default: 'o') and symbol_size: int (to define the symbol size)

Attributes**crosshair**

Convenience method

labels**roi_manager**

Convenience method

roi_target

To be implemented if necessary (Viewer1D and above)

Methods

<code>activate_roi([activate])</code>	Activate the Roi manager using the corresponding action
<code>move_roi_target([pos])</code>	move a specific read only ROI at the given position on the viewer
<code>set_crosshair_position(xpos[, ypos])</code>	Convenience method to set the crosshair positions

add_plot_item	
crosshair_changed	
double_clicked	
get_axis_from_view	
prepare_connect_ui	
process_crosshair_lineouts	
process_roi_lineouts	
roi_changed	
selected_region_changed	
update_colors	
update_status	

activate_roi(activate=True)

Activate the Roi manager using the corresponding action

move_roi_target(pos: Optional[Iterable[float]] = None)

move a specific read only ROI at the given position on the viewer

set_crosshair_position(xpos, ypos=0)

Convenience method to set the crosshair positions

property crosshair

Convenience method

property roi_manager

Convenience method

property roi_target: `InfiniteLine`

To be implemented if necessary (Viewer1D and above)

class pymodaq.utils.plotting.data_viewers.viewer2D.**Viewer2D**(parent: *Optional*[*QWidget*] = None, title="")

Object managing plotting and manipulation of 2D data using a View2D

Attributes

crosshair

Convenience method

image_widget

Convenience method

roi_manager

Convenience method

roi_target

To be implemented if necessary (Viewer1D and above)

x_axis

y_axis

Methods

<i>activate_roi</i> ([activate])	Activate the Roi manager using the corresponding action
<i>get_axes_from_view</i> (data)	Obtain axes info from the view
<i>get_data_at</i> ()	Convenience method
<i>move_roi_target</i> ([pos, size])	move a specific read only ROI at the given position on the viewer
<i>set_crosshair_position</i> (xpos, ypos)	Convenience method to set the crosshair positions
<i>set_gradient</i> (image_key, gradient)	convenience function
<i>set_image_transform</i> ()	Deactivate some tool buttons if data type is "spread" then apply transform_image
<i>show_roi</i> ([show, show_roi_widget])	convenience function to control roi

autolevels_first	
crosshair_changed	
double_clicked	
prepare_connect_ui	
process_crosshair_lineouts	
process_roi_lineouts	
roi_changed	
selected_region_changed	
set_visible_items	
transform_image	
update_crosshair_data	
update_data	

activate_roi(activate=True)

Activate the Roi manager using the corresponding action

get_axes_from_view(data: *DataWithAxes*)

Obtain axes info from the view

Only for uniform data

get_data_at()

Convenience method

move_roi_target(pos: *Optional[Iterable[float]]* = None, size: *Iterable[float]* = (1, 1))

move a specific read only ROI at the given position on the viewer

set_crosshair_position(xpos, ypos)

Convenience method to set the crosshair positions

set_gradient(image_key, gradient)

convenience function

set_image_transform() → *DataRaw*

Deactivate some tool buttons if data type is “spread” then apply transform_image

show_roi(show=True, show_roi_widget=True)

convenience function to control roi

property crosshair

Convenience method

property image_widget

Convenience method

property roi_manager

Convenience method

property roi_target: ROI

To be implemented if necessary (Viewer1D and above)

class pymodaq.utils.plotting.data_viewers.viewerND.**ViewerND**(parent: *Optional[QWidget]* = None, title=“”)

Methods

<i>setup_actions</i> ()	Method where to create actions to be subclassed.
-------------------------	--

connect_things	
prepare_ui	
reshape_data	
set_data_test	
setup_widgets	
show_settings	
update_data_dim	
update_data_displayer	
update_filters	
update_widget_visibility	

setup_actions()

Method where to create actions to be subclassed. Mandatory

Examples

```
>>> self.add_action('Quit', 'close2', "Quit program")
>>> self.add_action('Grab', 'camera', "Grab from camera", checkable=True)
>>> self.add_action('Load', 'Open', "Load target file (.h5, .png, .jpg) or _
↳data from camera", checkable=False)
>>> self.add_action('Save', 'SaveAs', "Save current data", checkable=False)
```

See also:

ActionManager.add_action

Plotting utility classes

class pymodaq.utils.plotting.scan_selector.**ScanSelector**(viewer_items: *Optional[List[SelectorItem]]*
= None, positions: *Optional[List] = None*)

Allows selection of a given 2D viewer to get scan info

respectively scan2D or scan Tabular from respectively a rectangular ROI or a polyline

Parameters

- **viewer_items** (*dict*) – where the keys are the titles of the sources while the values are dict with keys * viewers: list of plotitems * names: list of viewer titles
- **selector_type** (*str*) – either ‘PolyLines’ corresponding to a polyline ROI or ‘Rectangle’ for a rect Roi
- **positions** (*list*) – a sequence of 2 floats sequence [(x1,y1),(x2,y2),(x3,y3),...]

Attributes

selector_type
source_name
viewers_items

Methods

<i>value_changed</i> (param)	Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed
------------------------------	---

hide	
remove_selector	
scan_select_signal	
show	
show_selector	
update_model	
update_model_data	
update_scan	
update_selector_type	
update_table_view	

value_changed(*param*)

Non-mandatory method to be subclassed for actions to perform (methods to call) when one of the param's value in self._settings is changed

Parameters

param (*Parameter*) – the parameter whose value just changed

Examples

```
>>> if param.name() == 'do_something':
>>>     if param.value():
>>>         print('Do something')
>>>         self.settings.child('main_settings', 'something_done').
→setValue(False)
```

class pymodaq.utils.gui_utils.widgets.lcd.LCD(*parent: Optional[QObject] = None*)

Methods

<code>setvalues(values)</code>	display values on lcds :param values: :type values: list of 0D ndarray
--------------------------------	---

setupui	
---------	--

setvalues(*values: List[ndarray]*)

display values on lcds :param values: :type values: list of 0D ndarray

7.8.4 Utility Libraries

Utility Classes

class pymodaq.utils.daq_utils.ThreadCommand(*command: str, attribute=None, attributes=None*)

Generic object to pass info (command) and data (attribute) between thread or objects using signals

Parameters

- **command** (*str*) – The command to be analysed for further action
- **attribute** (*any type*) – the attribute related to the command. The actual type and value depend on the command and the situation
- **attributes** (*deprecated, attribute should be used instead*) –

command

The command to be analysed for further action

Type

str

attribute

the attribute related to the command. The actual type and value depend on the command and the situation

Type

any type

TCP/IP related methods**Serializing object to bytes and back**

Created the 20/10/2023

@author: Sebastien Weber

class pymodaq.utils.tcp_ip.serializer.**DeSerializer**(*bytes_string*: *Union[bytes, Socket]* = None)

Used to DeSerialize bytes to python objects, numpy arrays and PyMoDAQ Axis, DataWithAxes and DataToExport objects

Parameters

bytes_string (*bytes* or *Socket*) – the bytes string to deserialize into an object: int, float, string, arrays, list, Axis, DataWithAxes... Could also be a Socket object reading bytes from the network having a *get_first_nbytes* method

See also:

SocketString, *Socket*

Methods

<i>axis_deserialization()</i>	Convert bytes into an Axis object
<i>boolean_deserialization()</i>	Convert bytes into a boolean object
<i>bytes_to_int</i> (bytes_string)	Convert a bytes of length 4 into an integer
<i>bytes_to_nd_array</i> (data, dtype, shape)	Convert bytes to a ndarray given a certain numpy dtype and shape
<i>bytes_to_scalar</i> (data, dtype)	Convert bytes to a scalar given a certain numpy dtype
<i>dte_deserialization()</i>	Convert bytes into a DataToExport object
<i>dwa_deserialization()</i>	Convert bytes into a DataWithAxes object
<i>list_deserialization()</i>	Convert bytes into a list of homogeneous objects
<i>ndarray_deserialization()</i>	Convert bytes into a numpy ndarray object
<i>scalar_deserialization()</i>	Convert bytes into a numbers.Number object
<i>string_deserialization()</i>	Convert bytes into a str object

bytes_to_string	
from_b64_string	
object_deserialization	

axis_deserialization() → *Axis*

Convert bytes into an Axis object

Convert the first bytes into an Axis reading first information about the Axis

Returns

Axis

Return type

the decoded Axis

boolean_deserialization() → `bool`

Convert bytes into a boolean object

Get first the data type from a string deserialization, then the data length and finally convert this length of bytes into a number (float, int) and cast it to a bool

Returns`bool`**Return type**

the decoded boolean

static bytes_to_int(*bytes_string: bytes*) → `int`

Convert a bytes of length 4 into an integer

static bytes_to_nd_array(*data: bytes, dtype: dtype, shape: Tuple[int]*) → `ndarray`

Convert bytes to a ndarray given a certain numpy dtype and shape

Parameters

- **data** (*bytes*) –
- **dtype** (*np.dtype*) –
- **shape** (*tuple of int*) –

Return type`np.ndarray`**static bytes_to_scalar**(*data: bytes, dtype: dtype*) → `Number`

Convert bytes to a scalar given a certain numpy dtype

Parameters

- **data** (*bytes*) –
- **dtype** (*np.dtype*) –

Return type`numbers.Number`**dte_deserialization()** → `DataToExport`

Convert bytes into a DataToExport object

Convert the first bytes into a DataToExport reading first information about the underlying data

Returns`DataToExport`**Return type**

the decoded DataToExport

dwa_deserialization() → `DataWithAxes`

Convert bytes into a DataWithAxes object

Convert the first bytes into a DataWithAxes reading first information about the underlying data

Returns`DataWithAxes`**Return type**

the decoded DataWithAxes

list_deserialization() → *list*

Convert bytes into a list of homogeneous objects

Convert the first bytes into a list reading first information about the list elt types, length ...

Returns

list

Return type

the decoded list

ndarray_deserialization() → *ndarray*

Convert bytes into a numpy ndarray object

Convert the first bytes into a ndarray reading first information about the array's data

Returns

ndarray

Return type

the decoded numpy array

scalar_deserialization() → *Number*

Convert bytes into a numbers.Number object

Get first the data type from a string deserialization, then the data length and finally convert this length of bytes into a number (float, int)

Returns

numbers.Number

Return type

the decoded number

string_deserialization() → *str*

Convert bytes into a str object

Convert first the fourth first bytes into an int encoding the length of the string to decode

Returns

str

Return type

the decoded string

class pymodaq.utils.tcp_ip.serializer.**Serializer**(*obj: Optional[Union[int, str, Number, list, ndarray, Axis, DataWithAxes, DataToExport]] = None*)

Used to Serialize to bytes python objects, numpy arrays and PyMoDAQ DataWithAxes and DataToExport objects

Methods

<code>axis_serialization(axis)</code>	Convert an Axis object into a bytes message together with the info to convert it back
<code>dte_serialization(dte)</code>	Convert a DataToExport into a bytes string
<code>dwa_serialization(dwa)</code>	Convert a DataWithAxes into a bytes string
<code>int_to_bytes(an_integer)</code>	Convert an unsigned integer into a byte array of length 4 in big endian
<code>list_serialization(list_object)</code>	Convert a list of objects into a bytes message together with the info to convert it back
<code>ndarray_serialization(array)</code>	Convert a ndarray into a bytes message together with the info to convert it back
<code>object_type_serialization(obj)</code>	Convert an object type into a bytes message as a string together with the info to convert it back
<code>scalar_serialization(scalar)</code>	Convert a scalar into a bytes message together with the info to convert it back
<code>str_len_to_bytes(message)</code>	Convert a string and its length to two bytes :param message: the message to convert :type message: str
<code>string_serialization(string)</code>	Convert a string into a bytes message together with the info to convert it back
<code>to_bytes()</code>	Generic method to obtain the bytes string from various objects

bytes_serialization	
str_to_bytes	
to_b64_string	
type_and_object_serialization	

axis_serialization(axis: [Axis](#)) → [bytes](#)

Convert an Axis object into a bytes message together with the info to convert it back

Parameters

axis ([Axis](#)) –

Returns

bytes

Return type

the total bytes message to serialize the Axis

Notes

The bytes sequence is constructed as:

- serialize the type: 'Axis'
- serialize the axis label
- serialize the axis units
- serialize the axis array
- serialize the axis
- serialize the axis spread_order

dte_serialization(*dte*: *DataToExport*) → bytes

Convert a DataToExport into a bytes string

Parameters

dte (*DataToExport*) –

Returns

bytes

Return type

the total bytes message to serialize the DataToExport

Notes

The bytes sequence is constructed as:

- serialize the string type: 'DataToExport'
- serialize the timestamp: float
- serialize the name
- serialize the list of DataWithAxes

dwa_serialization(*dwa*: *DataWithAxes*) → bytes

Convert a DataWithAxes into a bytes string

Parameters

dwa (*DataWithAxes*) –

Returns

bytes

Return type

the total bytes message to serialize the DataWithAxes

Notes

The bytes sequence is constructed as:

- serialize the string type: 'DataWithAxes'
- serialize the timestamp: float
- serialize the name
- serialize the source enum as a string
- serialize the dim enum as a string
- serialize the distribution enum as a string
- serialize the list of numpy arrays
- serialize the list of labels
- serialize the origin
- serialize the nav_index tuple as a list of int
- serialize the list of axis
- serialize the errors attributes (None or list(np.ndarray))

- serialize the list of names of extra attributes
- serialize the extra attributes

static int_to_bytes(*an_integer*: *int*) → *bytes*

Convert an unsigned integer into a byte array of length 4 in big endian

Parameters

an_integer (*int*) –

Return type

bytearray

list_serialization(*list_object*: *List*) → *bytes*

Convert a list of objects into a bytes message together with the info to convert it back

Parameters

list_object (*list*) – the list could contains either scalars, strings or ndarrays or Axis objects or DataWithAxis objects module

Returns

bytes

Return type

the total bytes message to serialize the list of objects

Notes

The bytes sequence is constructed as: * the length of the list

Then for each object:

- get data type as a string
- use the serialization method adapted to each object in the list

ndarray_serialization(*array*: *ndarray*) → *bytes*

Convert a ndarray into a bytes message together with the info to convert it back

Parameters

array (*np.ndarray*) –

Returns

bytes

Return type

the total bytes message to serialize the scalar

Notes

The bytes sequence is constructed as:

- get data type as a string
- reshape array as 1D array and get the array dimensionality (len of array's shape)
- convert Data array as bytes
- serialize data type
- serialize data length

- serialize data shape length
- serialize all values of the shape as integers converted to bytes
- serialize array as bytes

object_type_serialization(*obj*: *Union*[*Axis*, *DataToExport*, *DataWithAxes*]) → *bytes*

Convert an object type into a bytes message as a string together with the info to convert it back

Applies to Data object from the pymodaq.utils.data module

scalar_serialization(*scalar*: *Number*) → *bytes*

Convert a scalar into a bytes message together with the info to convert it back

Parameters

scalar (*str*) –

Returns

bytes

Return type

the total bytes message to serialize the scalar

classmethod str_len_to_bytes(*message*: *~typing.Union*[*str*, *bytes*]) -> (<class 'bytes'>, <class 'bytes'>)

Convert a string and its length to two bytes :param message: the message to convert :type message: str

Returns

- *bytes* (*message converted as a byte array*)
- *bytes* (*length of the message byte array, itself as a byte array of length 4*)

string_serialization(*string*: *str*) → *bytes*

Convert a string into a bytes message together with the info to convert it back

Parameters

string (*str*) –

Returns

bytes

Return type

the total bytes message to serialize the string

to_bytes()

Generic method to obtain the bytes string from various objects

Compatible objects are:

- *bytes*
- *numbers.Number*
- *str*
- *numpy.ndarray*
- *Axis*
- *DataWithAxes* and sub-flavours
- *DataToExport*
- *list* of any objects above

class pymodaq.utils.tcp_ip.serializer.**SocketString**(*bytes_string: bytes*)

Mimic the Socket object but actually using a bytes string not a socket connection

Implements a minimal interface of two methods

Parameters

bytes_string (*bytes*) –

See also:

[Socket](#)

Methods

<i>check_received_length</i> (length)	Make sure all bytes (length) that should be received are received through the socket.
<i>get_first_nbytes</i> (length)	Read the first N bytes from the socket

check_received_length(*length: int*) → *bytes*

Make sure all bytes (length) that should be received are received through the socket.

Here just read the content of the underlying bytes string

Parameters

length (*int*) – The number of bytes to be read from the socket

Return type

bytes

get_first_nbytes(*length: int*) → *bytes*

Read the first N bytes from the socket

Parameters

length (*int*) – The number of bytes to be read from the socket

Returns

the read bytes string

Return type

bytes

Custom Sockets to implement PyMoDAQ protocol

Created the 26/10/2023

@author: Sebastien Weber

class pymodaq.utils.tcp_ip.mysocket.**Socket**(*socket: Optional[socket] = None*)

Custom Socket wrapping the built-in one and added functionalities to make sure message have been sent and received entirely

Attributes

socket

Methods

<code>check_received_length</code> (length)	Make sure all bytes (length) that should be received are received through the socket
<code>check_sended</code> (data_bytes)	Make sure all bytes are sent through the socket :param data_bytes: :type data_bytes: bytes
<code>check_sended_with_serializer</code> (obj)	Convenience function to convert permitted objects to bytes and then use the <code>check_sended</code> method
<code>get_first_nbytes</code> (length)	Read the first N bytes from the socket

accept	
bind	
close	
connect	
getsockname	
listen	
recv	
send	
sendall	

check_received_length(length) → bytes

Make sure all bytes (length) that should be received are received through the socket

Parameters

length (*int*) – The number of bytes to be read from the socket

Return type

bytes

check_sended(data_bytes: bytes)

Make sure all bytes are sent through the socket :param data_bytes: :type data_bytes: bytes

check_sended_with_serializer(obj: object)

Convenience function to convert permitted objects to bytes and then use the `check_sended` method

For a list of allowed objects, see `Serializer.to_bytes()`

get_first_nbytes(length: int) → bytes

Read the first N bytes from the socket

Parameters

length (*int*) – The number of bytes to be read from the socket

Returns

bytes

Return type

the read bytes string

Base classes as TCP server and client

Created on Fri Aug 30 12:21:56 2019

@author: Weber

class pymodaq.utils.tcp_ip.tcp_server_client.**Grabber**(parent: Optional[QObject] = None)

Methods

<code>grab_data()</code>	Do a grab session using 2 profile :
--------------------------	-------------------------------------

command_tcpip	
connect_tcp_ip	
process_tcpip_cmds	
snapshot	

grab_data()

Do a grab session using 2 profile :

- if grab pb checked do a continuous save and send an “update_channels” thread command and a “grab” too.
- if not send a “stop_grab” thread command with settings “main settings-naverage” node value as an attribute.

See also:

daq_utils.ThreadCommand, set_enabled_Ini_buttons

class pymodaq.utils.tcp_ip.tcp_server_client.**MockServer**(client_type='GRABBER')

class pymodaq.utils.tcp_ip.tcp_server_client.**TCPCClient**(ipaddress='192.168.1.62', port=6341,
params_state=None,
client_type='GRABBER')

PyQt5 object initializing a TCP socket client. Can be used by any module but is a builtin functionality of all actuators and detectors of PyMoDAQ

The module should init TCPCClient, move it in a thread and communicate with it using a custom signal connected to TCPCClient.queue_command slot. The module should also connect TCPCClient.cmd_signal to one of its methods in order to get info/data back from the client

The client itself communicate with a TCP server, it is best to use a server object subclassing the TCPServer class defined within this python module

Parameters

params_state ((dict) state of the Parameter settings of the module instantiating this client and wishing to) – export its settings to the server. Obtained from param.saveState() where param is an instance of Parameter object, see pyqtgraph.parametertree::Parameter

Methods

<code>get_data(message)</code>	param message
<code>post_init([extra_commands])</code>	To implement in a real object implementation
<code>queue_command([command])</code>	when this TCPClient object is within a thread, the corresponding module communicate with it with signal and slots from module to client: module_signal to queue_command slot from client to module: self.cmd_signal to a module slot
<code>ready_to_read()</code>	Do stuff (like read data) when messages arrive through the socket
<code>ready_to_write()</code>	Send stuff into the socket
<code>ready_with_error()</code>	Error in the socket communication

cmd_signal	
data_ready	
not_connected	
process_error_in_polling	
send_data	
send_info_string	
send_infos_xml	

get_data(*message: str*)

Parameters

message –

post_init(*extra_commands=[]*)

To implement in a real object implementation

queue_command(*command=<class 'pymodaq.utils.daq_utils.ThreadCommand'>*)

when this TCPClient object is within a thread, the corresponding module communicate with it with signal and slots from module to client: module_signal to queue_command slot from client to module: self.cmd_signal to a module slot

ready_to_read()

Do stuff (like read data) when messages arrive through the socket

ready_to_write()

Send stuff into the socket

ready_with_error()

Error in the socket communication

class pymodaq.utils.tcp_ip.tcp_server_client.**TCPServer**(*client_type='GRABBER'*)

Abstract class to be used as inherited by DAQ_Viewer_TCP or DAQ_Move_TCP

Methods

<code>close_server()</code>	close the current opened server.
<code>find_socket_type_within_connected_clients(sock)</code>	Find a socket type from a connected client with socket content corresponding.
<code>find_socket_within_connected_clients(client_type)</code>	Find a socket from a connected client with socket type corresponding.
<code>listen_client()</code>	Server function.
<code>print_status(status)</code>	Print the given status.
<code>process_cmds(command[, command_sock])</code>	Process the given command.
<code>read_info([sock, test_info, test_value])</code>	if the client is not from PyMoDAQ it can use this method to display some info into the server widget
<code>select(rlist[, wlist, xlist, timeout])</code>	Implements the select method, https://docs.python.org/3/library/select.html :param rlist: :type rlist: (list) wait until ready for reading :param wlist: :type wlist: (list) wait until ready for writing :param xlist: :type xlist: (list) wait for an “exceptional condition” :param timeout: When the timeout argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks. :type timeout: (float) optional timeout argument specifies a time-out as a floating point number in seconds.
<code>send_command(sock[, command])</code>	Send one of the message contained in self.message_list toward a socket with identity socket_type.
<code>timerEvent(event)</code>	Called by set timers.

command_done	
command_to_from_client	
emit_status	
init_server	
read_data	
read_info_xml	
read_infos	
remove_client	
send_data	
set_connected_clients_table	

`close_server()`

close the current opened server. Update the settings tree consequently.

See also:

`set_connected_clients_table`, `daq_utils.ThreadCommand`

`find_socket_type_within_connected_clients(sock)`

Find a socket type from a connected client with socket content corresponding.

Parameters	Type	Description
<code>sock</code>	???	The socket content corresponding.

Returns

the socket dictionary

Return type

dictionary

find_socket_within_connected_clients(*client_type*) → *Socket*

Find a socket from a connected client with socket type corresponding.

Parameters	Type	Description
<i>client_type</i>	string	The corresponding client type

Returns

the socket dictionary

Return type

dictionary

listen_client()

Server function. Used to connect or listen incoming message from a client.

print_status(*status*)

Print the given status.

Parameters	Type	Description
<i>status</i>	string list	a string list representing the status socket

process_cmds(*command*, *command_sock=None*)

Process the given command.

read_info(*sock*: *Optional*[*Socket*] = *None*, *test_info*='an_info', *test_value*='')

if the client is not from PyMoDAQ it can use this method to display some info into the server widget

select(*rlist*, *wlist=[]*, *xlist=[]*, *timeout=0*)

Implements the select method, <https://docs.python.org/3/library/select.html> :param rlist: :type rlist: (list) wait until ready for reading :param wlist: :type wlist: (list) wait until ready for writing :param xlist: :type xlist: (list) wait for an “exceptional condition” :param timeout: When the timeout argument is omitted the function blocks until at least one file descriptor is ready.

A time-out value of zero specifies a poll and never blocks.

Returns

- **list** (*readable sockets*)
- **list** (*writable sockets*)
- **list** (*sockets with error pending*)

send_command(*sock*: *Socket*, *command*='move_at')

Send one of the message contained in self.message_list toward a socket with identity socket_type. First send the length of the command with 4bytes.

Parameters	Type	Description
<i>sock</i>	???	The current socket
<i>command</i>	string	The command as a string

See also:

`utility_classes.DAQ_Viewer_base.emit_status,` `daq_utils.ThreadCommand,`
`message_to_bytes`

`set_connected_clients_table()`

`timerEvent(event)`

Called by set timers. If the process is free, start the `listen_client` function.

Parameters	Type	Description
<i>event</i>	QTimerEvent object	Containing id from timer issuing this event

See also:

`listen_client`

Units conversion

Created the 27/10/2022

@author: Sebastien Weber

`pymodaq.utils.units.Ecmrel2Enm(Ecmrel, ref_wavelength=515)`

Converts energy from cm-1 relative to a ref wavelength to an energy in wavelength (nm)

Parameters

- **Ecmrel** (*float*) – photon energy in cm-1
- **ref_wavelength** (*float*) – reference wavelength in nm from which calculate the photon relative energy

Returns

photon energy in nm

Return type

float

Examples

```
>>> Ecmrel2Enm(500, 515)
528.6117526302285
```

`pymodaq.utils.units.Enm2cmrel(E_nm, ref_wavelength=515)`

Converts energy in nm to cm-1 relative to a ref wavelength

Parameters

- **E_nm** (*float*) – photon energy in wavelength (nm)
- **ref_wavelength** (*float*) – reference wavelength in nm from which calculate the photon relative energy

Returns

photon energy in cm-1 relative to the ref wavelength

Return type

float

Examples

```
>>> Enm2cmrel(530, 515)
549.551199853453
```

`pymodaq.utils.units.cm2nm(E_cm)`

Converts photon energy from absolute cm-1 to wavelength

Parameters

E_cm (*float*) – photon energy in cm-1

Returns

Photon energy in nm

Return type

float

Examples

```
>>> cm2nm(1e5)
100
```

`pymodaq.utils.units.eV2cm(E_eV)`

Converts photon energy from electronvolt to absolute cm-1

Parameters

E_eV (*float*) – Photon energy in eV

Returns

photon energy in cm-1

Return type

float

Examples

```
>>> eV2cm(0.07)
564.5880342655984
```

`pymodaq.utils.units.eV2nm(E_eV)`

Converts photon energy from electronvolt to wavelength in nm

Parameters

E_eV (*float*) – Photon energy in eV

Returns

photon energy in nm

Return type

float

Examples

```
>>> eV2nm(1.55)
799.898112990037
```

`pymodaq.utils.units.l2w(x, speedlight=300)`

Converts photon energy in rad/fs to nm (and vice-versa)

Parameters

- **x** (*float*) – photon energy in wavelength or rad/fs
- **speedlight** (*float*, *optional*) – the speed of light, by default 300 nm/fs

Return type

float

Examples

```
>>> l2w(800)
2.356194490192345
>>> l2w(800, 3e8)
2356194.490192345
```

`pymodaq.utils.units.nm2cm(E_nm)`

Converts photon energy from wavelength to absolute cm-1

Parameters

E_nm (*float*) – Photon energy in nm

Returns

photon energy in cm-1

Return type

float

Examples

```
>>> nm2cm(0.04)
0.000025
```

`pymodaq.utils.units.nm2eV(E_nm)`

Converts photon energy from wavelength in nm to electronvolt

Parameters

E_nm (*float*) – Photon energy in nm

Returns

photon energy in eV

Return type

float

Examples

```
>>> nm2eV(800)
1.549802593918197
```

Mathematical utilities

`pymodaq.utils.math_utils.find_index(x, threshold: Union[Number, List[Number]]) → List[tuple]`

`find_index` finds the index `ix` such that `x[ix]` is the closest from `threshold`

Parameters

- **x** (*vector*) –
- **threshold** (*list of real numbers*) –

Returns

out – `out=[(ix0,xval0),(ix1,xval1),...]`

Return type

list of 2-tuple containing ix,x[ix]

`pymodaq.utils.math_utils.ft(x, dim=-1)`

Process the 1D fast fourier transform and swaps the axis to get coorect results using `ftAxis` :param `x`: :type `x`: (ndarray) the array on which the FFT should be done :param `dim`: :type `dim`: the axis over which is done the FFT (default is the last of the array)

See also:

`ftAxis`, `ftAxis_time`, `ift`, `ft2`, `ift2`

`pymodaq.utils.math_utils.ft2(x, dim=(-2, -1))`

Process the 2D fast fourier transform and swaps the axis to get correct results using `ftAxis` :param `x`: :type `x`: (ndarray) the array on which the FFT should be done :param `dim`: :type `dim`: the axis over which is done the FFT (default is the last of the array)

See also:

`ftAxis`, `ftAxis_time`, `ift`, `ft2`, `ift2`

`pymodaq.utils.math_utils.ftAxis(Npts, omega_max)`

Given two numbers `Npts`,`omega_max`, return two vectors spanning the temporal and spectral range. They are related by Fourier Transform

Parameters

- **Npts** (*(int)*) – A number of points defining the length of both grids
- **omega_max** (*(float)*) – The maximum circular frequency in the spectral domain. its unit defines the temporal units. ex: `omega_max` in rad/fs implies `time_grid` in fs

Returns

- **omega_grid** (*(ndarray)*) – The spectral axis of the FFT
- **time_grid** (*(ndarray)*) – The temporal axis of the FFT

See also:

`ftAxis`, `ftAxis_time`, `ift`, `ft2`, `ift2`

`pymodaq.utils.math_utils.ftAxis_time(Npts, time_max)`

Given two numbers `Npts`, `omega_max`, return two vectors spanning the temporal and spectral range. They are related by Fourier Transform

Parameters

- **Npts** (*number*) – A number of points defining the length of both grids
- **time_max** (*number*) – The maximum temporal window

Returns

- **omega_grid** (*vector*) – The spectral axis of the FFT
- **time_grid** (*vector*) – The temporal axis of the FFT

See also:

[`ftAxis`](#), [`ftAxis_time`](#), [`ift`](#), [`ft2`](#), [`ift2`](#)

`pymodaq.utils.math_utils.gauss1D(x, x0, dx, n=1)`

compute the gaussian function along a vector `x`, centered in `x0` and with a FWHM i intensity of `dx`. `n=1` is for the standart gaussian while `n>1` defines a hypergaussian

Parameters

- **x** (*ndarray*) *first axis of the 2D gaussian* –
- **x0** (*float*) *the central position of the gaussian* –
- **dx** (*float*) *:the FWHM of the gaussian* –
- **n=1** *(an integer to define hypergaussian, n=1 by default for regular gaussian)* –

Returns

out – the value taken by the gaussian along x axis

Return type

vector

`pymodaq.utils.math_utils.gauss2D(x, x0, dx, y, y0, dy, n=1, angle=0)`

compute the 2D gaussian function along a vector `x`, centered in `x0` and with a FWHM in intensity of `dx` and `smay` along y axis. `n=1` is for the standard gaussian while `n>1` defines a hypergaussian. optionally rotate it by an angle in degree

Parameters

- **x** (*ndarray*) *first axis of the 2D gaussian* –
- **x0** (*float*) *the central position of the gaussian* –
- **dx** (*float*) *:the FWHM of the gaussian* –
- **y** (*ndarray*) *second axis of the 2D gaussian* –
- **y0** (*float*) *the central position of the gaussian* –
- **dy** (*float*) *:the FWHM of the gaussian* –
- **n=1** *(an integer to define hypergaussian, n=1 by default for regular gaussian)* –
- **angle** (*float*) *a float to rotate main axes, in degree* –

Returns

out

Return type

ndarray 2 dimensions

`pymodaq.utils.math_utils.ift(x, dim=0)`

Process the inverse 1D fast fourier transform and swaps the axis to get correct results using ftAxis :param x: :type x: (ndarray) the array on which the FFT should be done :param dim: :type dim: the axis over which is done the FFT (default is the last of the array)

See also:[*ftAxis*](#), [*ftAxis_time*](#), [*ift*](#), [*ft2*](#), [*ift2*](#)`pymodaq.utils.math_utils.ift2(x, dim=(-2, -1))`

Process the inverse 2D fast fourier transform and swaps the axis to get correct results using ftAxis :param x: :type x: (ndarray) the array on which the FFT should be done :param dim: :type dim: the axis (or a tuple of axes) over which is done the FFT (default is the last of the array)

See also:[*ftAxis*](#), [*ftAxis_time*](#), [*ift*](#), [*ft2*](#), [*ift2*](#)`pymodaq.utils.math_utils.linspace_step(start, stop, step)`

Compute a regular linspace_step distribution from start to stop values.

Parameters	Type	Description
<i>start</i>	scalar	the starting value of distribution
<i>stop</i>	scalar	the stopping value of distribution
<i>step</i>	scalar	the length of a distribution step

Returns

The computed distribution axis as an array.

Return type

scalar array

`pymodaq.utils.math_utils.my_moment(x, y)`

Returns the moments of a distribution y over an axe x

Parameters

- **x** ([*list*](#) or *ndarray*) – vector of floats
- **y** ([*list*](#) or *ndarray*) – vector of floats corresponding to the x axis

Returns

m – Contains moment of order 0 (mean) and of order 1 (std) of the distribution y

Return type[*list*](#)`pymodaq.utils.math_utils.odd_even(x)`

odd_even tells if a number is odd (return True) or even (return False)

Parameters

x (*the integer number to test*) –

Returns**bool****Return type**

boolean

Scan utilities

File management

`pymodaq.utils.gui_utils.select_file(start_path='C:\\Data', save=True, ext=None, filter=None, force_save_extension=False)`

Opens a selection file popup for loading or saving a file

Parameters

- **start_path** (*str* or *Path*) – The starting point in the file/folder system to open the popup from
- **save** (*bool*) – if True, ask you to enter a filename (with or without extension)
- **ext** (*str*) – the extension string, e.g. xml, h5, png ...
- **filter** (*string*) – list of possible extensions, if you need several you can separate them by ;; for example: “Images (.png *.xpm *.jpg);;Text files (.txt);;XML files (*.xml)”
- **force_save_extension** (*bool*) – if True force the extension of the saved file to be set to ext

Returns

Path

Return type

the Path object of the file to save or load

Data Management

<code>DataDim(value)</code>	Enum for dimensionality representation of data
<code>DataSource(value)</code>	Enum for source of data
<code>DataDistribution(value)</code>	Enum for distribution of data
<code>Axis([label, units, data, index, scaling, ...])</code>	Object holding info and data about physical axis of some data
<code>DataBase(name[, source, dim, distribution, ...])</code>	Base object to store homogeneous data and metadata generated by pymodaq's objects.
<code>DataRaw(*args, **kwargs)</code>	Specialized <code>DataWithAxes</code> set with source as 'raw'.
<code>DataCalculated(*args[, axes])</code>	Specialized <code>DataWithAxes</code> set with source as 'calculated'.
<code>DataFromPlugins(*args, **kwargs)</code>	Specialized <code>DataWithAxes</code> set with source as 'raw'.
<code>DataFromRoi(*args[, axes])</code>	Specialized <code>DataWithAxes</code> set with source as 'calculated'. To be used for processed data from region of interest
<code>DataToExport(name[, data])</code>	Object to store all raw and calculated <code>DataWithAxes</code> data for later exporting, saving, sending signal...

Axes

Created the 28/10/2022

@author: Sebastien Weber

```
class pymodaq.utils.data.Axis(label: str = "", units: str = "", data: Optional[ndarray] = None, index: int = 0,
                               scaling=None, offset=None, size=None, spread_order: int = 0)
```

Object holding info and data about physical axis of some data

In case the axis's data is linear, store the info as a scale and offset else store the data

Parameters

- **label** (*str*) – The label of the axis, for instance ‘time’ for a temporal axis
- **units** (*str*) – The units of the data in the object, for instance ‘s’ for seconds
- **data** (*ndarray*) – A 1D ndarray holding the data of the axis
- **index** (*int*) – an integer representing the index of the Data object this axis is related to
- **scaling** (*float*) – The scaling to apply to a linspace version in order to obtain the proper scaling
- **offset** (*float*) – The offset to apply to a linspace/scaled version in order to obtain the proper axis
- **size** (*int*) – The size of the axis array (to be specified if data is None)
- **spread_order** (*int*) – An integer needed in the case where data has a spread DataDistribution. It refers to the index along the data's spread_index dimension

Examples

```
>>> axis = Axis('myaxis', units='seconds', data=np.array([1,2,3,4,5]), index=0)
```

create_linear_data(*nsteps: int*)

replace the axis data with a linear version using scaling and offset

find_index(*threshold: float*) → *int*

find the index of the threshold value within the axis

get_data() → *ndarray*

Convenience method to obtain the axis data (usually None because scaling and offset are used)

get_data_at(*indexes: Union[int, Iterable, slice]*) → *ndarray*

Get data at specified indexes

Parameters

indexes –

get_scale_offset_from_data(*data: Optional[ndarray] = None*)

Get the scaling and offset from the axis's data

If data is not None, extract the scaling and offset

Parameters

data (*ndarray*) –

property data

get/set the data of Axis

Type

np.ndarray

property index: `int`

get/set the index this axis corresponds to in a DataWithAxis object

Type`int`**property label:** `str`

get/set the label of this axis

Type`str`**property size:** `int`

get/set the size/length of the 1D ndarray

Type`int`**property units:** `str`

get/set the units for this axis

Type`str`**DataObjects**

Created the 28/10/2022

@author: Sebastien Weber

```
class pymodaq.utils.data.DataBase(name: str, source: Optional[DataSource] = None, dim:
    Optional[DataDim] = None, distribution: DataDistribution =
    DataDistribution.uniform, data: Optional[List[ndarray]] = None,
    labels: Optional[List[str]] = None, origin: str = "", **kwargs)
```

Base object to store homogeneous data and metadata generated by pymodaq's objects.

To be inherited for real data

Parameters

- **name** (`str`) – the identifier of these data
- **source** (`DataSource` or `str`) – Enum specifying if data are raw or processed (for instance from roi)
- **dim** (`DataDim` or `str`) – The identifier of the data type
- **distribution** (`DataDistribution` or `str`) – The distribution type of the data: uniform if distributed on a regular grid or spread if on specific unordered points
- **data** (`list` of `ndarray`) – The data the object is storing
- **labels** (`list` of `str`) – The labels of the data nd-arrays

- **origin** (*str*) – An identifier of the element where the data originated, for instance the DAQ_Viewer’s name. Used when appending DataToExport in DAQ_Scan to disintricate from which origin data comes from when scanning multiple detectors.
- **kwargs** (*named parameters*) – All other parameters are stored dynamically using the name/value pair. The name of these extra parameters are added into the extra_attributes attribute

name

the identifier of these data

Type

str

source

Enum specifying if data are raw or processed (for instance from roi)

Type

DataSource or *str*

dim

The identifier of the data type

Type

DataDim or *str*

distribution

The distribution type of the data: uniform if distributed on a regular grid or spread if on specific unordered points

Type

DataDistribution or *str*

data

The data the object is storing

Type

list of *ndarray*

labels

The labels of the data nd-arrays

Type

list of *str*

origin

An identifier of the element where the data originated, for instance the DAQ_Viewer’s name. Used when appending DataToExport in DAQ_Scan to disintricate from which origin data comes from when scanning multiple detectors.

Type

str

shape

The shape of the underlying data

Type

Tuple[*int*]

size

The size of the ndarrays stored in the object

Type

int

length

The number of ndarrays stored in the object

Type

int

extra_attributes

list of string giving identifiers of the attributes added dynamically at the initialization (for instance to save extra metadata using the DataSaverLoader

Type

List[str]

See also:

DataWithAxes, [DataFromPlugins](#), [DataRaw](#), DataSaverLoader

Examples

```
>>> import numpy as np
>>> from pymodaq.utils.data import DataBase, DataSource, DataDim, DataDistribution
>>> data = DataBase('mydata', source=DataSource['raw'], dim=DataDim['Data1D'],
↵distribution=DataDistribution['uniform'], data=[np.array([1.,2.,3.]), np.array([4.
↵,5.,6.])], labels=['channel1', 'channel2'], origin='docutils code')
>>> data.dim
<DataDim.Data1D: 1>
>>> data.source
<DataSource.raw: 0>
>>> data.shape
(3,)
>>> data.length
2
>>> data.size
3
```

abs()

Take the absolute value of itself

as_dte(name: str = 'mydte') → DataToExport

Convenience method to wrap the DataWithAxes object into a DataToExport

average(other: DataBase, weight: int) → DataBase

Compute the weighted average between self and other DataBase

Parameters

- **other_data** (DataBase) –
- **weight** (int) – The weight the ‘other’ holds with respect to self

Returns

DataBase

Return type

the averaged DataBase object

fliplr()

Reverse the order of elements along axis 1 (left/right)

flipud()

Reverse the order of elements along axis 0 (up/down)

get_data_index(*index: int = 0*) → ndarray

Get the data by its index in the list, same as self[index]

get_dim_from_data(*data: List[ndarray]*)

Get the dimensionality DataDim from data

get_full_name() → str

Get the data full name including the origin attribute into the returned value

Returns

str

Return type

the name of the ataWithAxes data constructed as : origin/name

Examples

```
d0 = DataBase(name='datafromdet0', origin='det0')
```

imag()

Take the imaginary part of itself

pop(*index: int*) → *DataBase*

Returns a copy of self but with data taken at the specified index

real()

Take the real part of itself

set_dim(*dim: Union[DataDim, str]*)

Adhoc modification of dim independantly of the real data shape, should be used with extra care

stack_as_array(*axis=0, dtype=None*) → ndarray

Stack all data arrays in a single numpy array

Parameters

- **axis** (*int*) – The new stack axis index, default 0
- **dtype** (*str* or *np.dtype*) – the dtype of the stacked array

Return type

np.ndarray

See also:

```
np.stack()
```

property data: *List[ndarray]*

get/set (and check) the data the object is storing

Type*List[np.ndarray]*

property dim

the enum representing the dimensionality of the stored data

Type

DataDim

property distribution

the enum representing the distribution of the stored data

Type

DataDistribution

property length

The length of data. This is the length of the list containing the nd-arrays

property shape

The shape of the nd-arrays

property size

The size of the nd-arrays

property source

the enum representing the source of the data

Type

DataSource

class pymodaq.utils.data.**DataCalculated**(*args, axes=[], **kwargs)

Specialized DataWithAxes set with source as 'calculated'. To be used for processed/calculated data

class pymodaq.utils.data.**DataFromPlugins**(*args, **kwargs)

Specialized DataWithAxes set with source as 'raw'. To be used for raw data generated by Detector plugins

It introduces by default to extra attributes, do_plot and do_save. Their presence can be checked in the extra_attributes list.

Parameters

- **do_plot** (*bool*) – If True the underlying data will be plotted in the DAQViewer
- **do_save** (*bool*) – If True the underlying data will be saved

do_plot

If True the underlying data will be plotted in the DAQViewer

Type

bool

do_save

If True the underlying data will be saved

Type

bool

class pymodaq.utils.data.**DataFromRoi**(*args, axes=[], **kwargs)

Specialized DataWithAxes set with source as 'calculated'. To be used for processed data from region of interest

class pymodaq.utils.data.**DataRaw**(*args, **kwargs)

Specialized DataWithAxes set with source as 'raw'. To be used for raw data

Data Characteristics

Created the 28/10/2022

@author: Sebastien Weber

class pymodaq.utils.data.**DataDim**(*value*)

Enum for dimensionality representation of data

class pymodaq.utils.data.**DataDistribution**(*value*)

Enum for distribution of data

class pymodaq.utils.data.**DataSource**(*value*)

Enum for source of data

Union of Data

When exporting multiple set of Data objects, one should use a DataToExport

Created the 28/10/2022

@author: Sebastien Weber

class pymodaq.utils.data.**DataToExport**(*name: str, data: List[DataWithAxes] = [], **kwargs*)

Object to store all raw and calculated DataWithAxes data for later exporting, saving, sending signal...

Includes methods to retrieve data from dim, source... Stored data have a unique identifier their name. If some data is appended with an existing name, it will replace the existing data. So if you want to append data that has the same name

Parameters

- **name** (*str*) – The identifier of the exporting object
- **data** (*list of DataWithAxes*) – All the raw and calculated data to be exported

name

timestamp

data

affect_name_to_origin_if_none()

Affect self.name to all DataWithAxes children's attribute origin if this origin is not defined

average(*other: DataToExport, weight: int*) → *DataToExport*

Compute the weighted average between self and other DataToExport and attributes it to self

Parameters

- **other** (*DataToExport*) –
- **weight** (*int*) – The weight the 'other_data' holds with respect to self

get_data_from_Naxes(*Naxes: int, deepcopy: bool = False*) → *DataToExport*

Get the data matching the given number of axes

Parameters

- **Naxes** (*int*) – Number of axes in the DataWithAxes objects

Returns**DataToExport****Return type**

filtered with data matching the number of axes

get_data_from_attribute(*attribute*: *str*, *attribute_value*: *Any*, *deepcopy*=*False*) → *DataToExport*

Get the data matching a given attribute value

Returns**DataToExport****Return type**

filtered with data matching the attribute presence and value

get_data_from_dim(*dim*: *DataDim*, *deepcopy*=*False*) → *DataToExport*

Get the data matching the given DataDim

Returns**DataToExport****Return type**

filtered with data matching the dimensionality

get_data_from_dims(*dims*: *List*[*DataDim*], *deepcopy*=*False*) → *DataToExport*

Get the data matching the given DataDim

Returns**DataToExport****Return type**

filtered with data matching the dimensionality

get_data_from_full_name(*full_name*: *str*, *deepcopy*=*False*) → *DataWithAxes*

Get the DataWithAxes with matching full name

get_data_from_missing_attribute(*attribute*: *str*, *deepcopy*=*False*) → *DataToExport*

Get the data matching a given attribute value

Parameters

- **attribute** (*str*) – a string of a possible attribute
- **deepcopy** (*bool*) – if True the returned DataToExport will contain deepcopies of the DataWithAxes

Returns**DataToExport****Return type**

filtered with data missing the given attribute

get_data_from_name(*name*: *str*) → *DataWithAxes*

Get the data matching the given name

get_data_from_name_origin(*name*: *str*, *origin*: *str* = "") → *DataWithAxes*

Get the data matching the given name and the given origin

get_data_from_sig_axes(*Naxes*: *int*, *deepcopy*: *bool* = *False*) → *DataToExport*

Get the data matching the given number of signal axes

Parameters**Naxes** (*int*) – Number of signal axes in the DataWithAxes objects

Returns**DataToExport****Return type**

filtered with data matching the number of signal axes

get_data_from_source(*source*: [DataSource](#), *deepcopy*=False) → [DataToExport](#)

Get the data matching the given DataSource

Returns**DataToExport****Return type**

filtered with data matching the dimensionality

get_data_with_naxes_lower_than(*n_axes*=2, *deepcopy*: *bool* = False) → [DataToExport](#)

Get the data with n axes lower than the given number

Parameters**Naxes** (*int*) – Number of axes in the DataWithAxes objects**Returns****DataToExport****Return type**

filtered with data matching the number of axes

get_full_names(*dim*: *Optional*[[DataDim](#)] = None)

Get the full names including the origin attribute into the returned value, eventually filtered by dim

Parameters**dim** ([DataDim](#) or *str*) –**Returns**list of *str***Return type**

the names of the (filtered) DataWithAxes data constructed as : origin/name

Examples

```
d0 = DataWithAxes(name='datafromdet0', origin='det0')
```

get_names(*dim*: *Optional*[[DataDim](#)] = None) → [List](#)[*str*]

Get the names of the stored DataWithAxes, eventually filtered by dim

Parameters**dim** ([DataDim](#) or *str*) –**Returns**list of *str***Return type**

the names of the (filtered) DataWithAxes data

get_origins(*dim*: *Optional*[[DataDim](#)] = None)

Get the origins of the underlying data into the returned value, eventually filtered by dim

Parameters**dim** ([DataDim](#) or *str*) –

Returns

list of str

Return type

the origins of the (filtered) DataWithAxes data

Examples

```
d0 = DataWithAxes(name='datafromdet0', origin='det0')
```

```
index_from_name_origin(name: str, origin: str = "") → List[DataWithAxes]
```

Get the index of a given DataWithAxes within the list of data

```
merge_as_dwa(dim: Union[str, DataDim], name: Optional[str] = None) → DataRaw
```

attempt to merge filtered dwa into one

Only possible if all filtered dwa and underlying data have same shape

Parameters

- **dim** (*DataDim* or *str*) – will only try to merge dwa having this dimensionality
- **name** (*str*) – The new name of the returned dwa

```
plot(plotter_backend: str = 'matplotlib', *args, **kwargs)
```

Call a plotter factory and its plot method over the actual data

```
pop(index: int) → DataWithAxes
```

return and remove the DataWithAxes referred by its index

Parameters**index** (*int*) – index as returned by self.index_from_name_origin**See also:**[*index_from_name_origin*](#)

```
property data: List[DataWithAxes]
```

get the data contained in the object

Type

List[DataWithAxes]

parameter

Extension of the *pyqtgraph* Parameter, ParameterTree widgets and dedicated functions to deals with Parameters (e.g. save them in XML)

New Tree items

Documentation on the added or modified ParameterItem types compared to *pyqtgraph.parameterTree.parameterTypes* module.

WidgetParameterItem and SimpleParameter have been subclassed to define more options:

- int and float: represented by a custom Spinbox, see Spinbox
- bool, led, bool_push are represented respectively by a QCheckBox, a QLED, QPushButton

- `str` displays a `QLineEdit` widget
- `date_time` displays a `QDateTime` widget
- `date` displays a `QDate` widget
- `time` displays a `QTimeCustom` widget
- `pixmap` displays a `QPixmap` in a `QLabel`
- `pixmap_check` displays a custom `PixmapCheckWidget` widget

Other widgets for `ParameterTree` have been introduced:

- `group`: subclassed group parameter, see `GroupParameterCustom` and `GroupParameterItemCustom`
- `slide`: displays a custom `Spinbox` and a `QSlider` to set floats and ints, see `SliderSpinBox`
- `list`: subclassed `pyqtgraph` list that displays a list and a pushbutton to let the user add entries in the list, see `ListParameter` and `ListParameterItem` and `Combo_pb`
- `table`: subclassed `pyqtgraph` table, see `TableParameterItem`, `TableParameter` and `TableWidget`
- `table_view`: displaying a `QTableView` with custom model to be user defined, see Qt5 documentation, see `TableViewParameterItem`, `TableViewCustom` and `TableViewParameter`
- `itemselect`: displays a `QListWidget` with selectable elements, see `ItemSelectParameterItem`, `ItemSelect_pb`, `ItemSelect` and `ItemSelectParameter`
- `browsepath`: displays an edit line and a push button to select files or folders, see `FileDirParameterItem`, `FileDirWidget` and `FileDirParameter`
- `text``: subclassed plain text area `text` from `pyqtgraph` with limited height, see `TextParameterItemCustom` and `TextParameter`
- `text_pb`: displays a plain text area and a visible button to add data into it, see `PlainTextParameterItem`, `PlainTextWidget` and `PlainTextPbParameter`

Parameter and XML

Within PyMoDAQ, Parameter state are often saved or transferred (for instance when using TCP/IP) as a XML string whose Tree structure is well adapted to represent the Parameter tree structure. Below are all the functions used to convert from a Parameter to a XML string (or file) and vice-versa.

`pymodaq.utils.parameter.ioxml.XML_file_to_parameter(file_name: Union[str, Path]) → list`

Convert a xml file into `pyqtgraph` parameter object.

Returns

params – a list of dictionary defining a Parameter object and its children

Return type

list of dictionary

See also:

[`walk_parameters_to_xml`](#)

Examples

`pymodaq.utils.parameter.ioxml.XML_string_to_parameter(xml_string)`

Convert a xml string into a list of dict for initialize pyqtgraph parameter object.

Parameters	Type	Description
xml_string	string	the xml string to be converted

Returns

params

Return type

a parameter list of dict to init a parameter

See also:

[`walk_parameters_to_xml`](#)

Examples

`pymodaq.utils.parameter.ioxml.add_text_to_elt(elt, param)`

Add a text filed in a xml element corresponding to the parameter value

Parameters

- **elt** (*XML elt*) –
- **param** (*Parameter*) –

See also:

[`add_text_to_elt`](#), [`walk_parameters_to_xml`](#), [`dict_from_param`](#)

`pymodaq.utils.parameter.ioxml.dict_from_param(param)`

Get Parameter properties as a dictionary

Parameters

param (*Parameter*) –

Returns

opts

Return type

`dict`

See also:

[`add_text_to_elt`](#), [`walk_parameters_to_xml`](#), [`dict_from_param`](#)

`pymodaq.utils.parameter.ioxml.elt_to_dict(el)`

Convert xml element attributes to a dictionary

Parameters

el –

`pymodaq.utils.parameter.ioxml.parameter_to_xml_file(param, filename: Union[str, Path])`

Convert the given parameter to XML element and update the given XML file.

Parameters	Type	Description
<i>param</i>	instance of pyqtgraph parameter	the parameter to be added
<i>filename</i>	string	the filename of the XML file

See also:

[`walk_parameters_to_xml`](#)

Examples

`pymodaq.utils.parameter.ioxml.parameter_to_xml_string(param)`

Convert a Parameter to a XML string.

Parameters

param (*Parameter*) –

Returns

str

Return type

XML string

See also:

[`add_text_to_elt`](#), [`walk_parameters_to_xml`](#), [`dict_from_param`](#)

Examples

```
>>> from pyqtgraph.parametertree import Parameter
>>> #Create an instance of Parameter
>>> settings=Parameter(name='settings')
>>> converted_xml=parameter_to_xml_string(settings)
>>> # The converted Parameter
>>> print(converted_xml)
b'<settings title="settings" type="None" />'
```

`pymodaq.utils.parameter.ioxml.set_txt_from_elt(el, param_dict)`

get the value of the parameter from the text value of the xml element :param el: :type el: xml element :param param_dict: :type param_dict: dictionnary from which the parameter will be constructed

`pymodaq.utils.parameter.ioxml.walk_parameters_to_xml(parent_elt=None, param=None)`

To convert a parameter object (and children) to xml data tree.

Parameters	Type	Description
<i>parent_elt</i>	XML element	the root element
<i>param</i>	instance of pyqtgraph parameter	Parameter object to be converted

Returns

XML element – XML element with subelements from Parameter object

Return type

parent_elt

See also:[add_text_to_elt](#), [walk_parameters_to_xml](#), [dict_from_param](#)`pymodaq.utils.parameter.ioxml.walk_xml_to_parameter(params=[], XML_elt=None)`

To convert an XML element (and children) to list of dict enabling creation of parameter object.

Parameters	Type	Description
<i>params</i>	dictionary list	the list to create parameter object
<i>XML_elt</i>	XML object	the XML object to be converted

Returns**params** – list of dict to create parameter object**Return type**

dictionary list

Examples

```
>>> from pyqtgraph.parametertree import Parameter, ParameterItem
>>> import xml.etree.ElementTree as ET
>>> tree = ET.parse('text_bis.xml')
>>> root = tree.getroot()
>>> params=walk_xml_to_parameter(XML_elt=root)
>>> settings_xml=Parameter.create(name='Settings XML', type='group',
↳ children=params)
>>> settings=Parameter.create(name='Settings', type='group', children=params)
```

See also:[walk_parameters_to_xml](#)**Parameter management**

Utility functions to work with Parameter object

`pymodaq.utils.parameter.utils.get_param_from_name(parent, name) → Parameter`

Get Parameter under parent whose name is name

Parameters

- **parent** (*Parameter*) –
- **name** (*str*) –

Return type

Parameter

`pymodaq.utils.parameter.utils.get_param_path(param)`**Parameters****param** –

`pymodaq.utils.parameter.utils.iter_children(param, childlist=[])`

Get a list of parameters name under a given Parameter | Returns all childrens names.

Parameters	Type	Description
<i>param</i>	instance of pyqtgraph parameter	the root node to be coursed
<i>childlist</i>	list	the child list recetion structure

Returns

childlist – The list of the children from the given node.

Return type

parameter list

`pymodaq.utils.parameter.utils.iter_children_params(param, childlist=[])`

Get a list of parameters under a given Parameter

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

- `pymodaq.utils.daq_utils`, 311
- `pymodaq.utils.data`, 338
- `pymodaq.utils.gui_utils`, 331
- `pymodaq.utils.h5modules.browsing`, 289
- `pymodaq.utils.h5modules.data_saving`, 285
- `pymodaq.utils.h5modules.module_saving`, 291
- `pymodaq.utils.h5modules.saving`, 279
- `pymodaq.utils.math_utils`, 328
- `pymodaq.utils.parameter.ioxml`, 342
- `pymodaq.utils.parameter.pymodaq_ptypes`, 342
- `pymodaq.utils.parameter.utils`, 345
- `pymodaq.utils.scanner`, 331
- `pymodaq.utils.tcp_ip.mysocket`, 319
- `pymodaq.utils.tcp_ip.serializer`, 312
- `pymodaq.utils.tcp_ip.tcp_server_client`, 321
- `pymodaq.utils.units`, 325

A

- `abs()` (*pymodaq.utils.data.DataBase* method), 335
- `ActionManager` (class in *py-modaq.utils.managers.action_manager*), 296
- `activate_roi()` (*pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D* method), 307
- `activate_roi()` (*pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D* method), 308
- `Actuator`, 261
- `actuators` (*pymodaq.utils.managers.modules_manager.ModulesManager* property), 305
- `actuators` (*pymodaq.utils.scanner.Scanner* property), 295
- `actuators_all` (*pymodaq.utils.managers.modules_manager.ModulesManager* property), 305
- `actuators_name` (*pymodaq.utils.managers.modules_manager.ModulesManager* property), 305
- `ActuatorSaver` (class in *py-modaq.utils.h5modules.module_saving*), 291
- `add_action()` (*pymodaq.utils.managers.action_manager.ActionManager* method), 297
- `add_axis()` (*pymodaq.utils.h5modules.data_saving.AxisSaverLoader* method), 282
- `add_bkg()` (*pymodaq.utils.h5modules.module_saving.DetectorSaver* method), 291
- `add_comments()` (*pymodaq.utils.h5modules.browsing.H5Browser* method), 290
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataEnlargeableSaver* method), 285
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataExtendedSaver* method), 286
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataSaverLoader* method), 283
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataToExportEnlargeableSaver* method), 286
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataToExportExtendedSaver* method), 287
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataToExportSaver* method), 284
- `add_data()` (*pymodaq.utils.h5modules.data_saving.DataToExportTimedSaver* method), 288
- `add_data()` (*pymodaq.utils.h5modules.module_saving.LoggerSaver* method), 292
- `add_group()` (*pymodaq.utils.h5modules.backends.H5Backend* method), 277
- `add_max_axes()` (*pymodaq.utils.h5modules.data_saving.DataToExportExtendedSaver* method), 287
- `add_text_to_slt()` (in module *py-modaq.utils.parameter.ioxml*), 343
- `add_widget()` (*pymodaq.utils.managers.action_manager.ActionManager* method), 298
- `addaction()` (in module *py-modaq.utils.managers.action_manager*), 296
- `affect_name_to_origin_if_none()` (*py-modaq.utils.data.DataToExport* method), 338
- `affect_to()` (*pymodaq.utils.managers.action_manager.ActionManager* method), 298
- `as_dte()` (*pymodaq.utils.data.DataBase* method), 335
- `attribute` (*pymodaq.utils.daq_utils.ThreadCommand* attribute), 311
- `average()` (*pymodaq.utils.data.DataBase* method), 335
- `average()` (*pymodaq.utils.data.DataToExport* method), 338
- `Axis` (class in *pymodaq.utils.data*), 332
- `axis_deserialization()` (*py-modaq.utils.tcp_ip.serializer.DeSerializer* method), 312
- `axis_name` (*pymodaq.control_modules.move_utility_classes.DAQ_Move_b* property), 267
- `axis_names` (*pymodaq.control_modules.move_utility_classes.DAQ_Move_b* property), 267
- `axis_serialization()` (*py-modaq.utils.tcp_ip.serializer.Serializer* method), 315
- `axis_value` (*pymodaq.control_modules.move_utility_classes.DAQ_Move_b* property), 268
- `AxisSaverLoader` (class in *py-modaq.utils.h5modules.data_saving*), 282

B

`BayesianModelDefault` (class in `pymodaq.extensions`), 272

`BayesianModelGeneric` (class in `pymodaq.extensions`), 271

`BayesianOptimisation` (class in `pymodaq.extensions`), 268

`BkgSaver` (class in `py-modaq.utils.h5modules.data_saving`), 285

`boolean_deserialization()` (py-modaq.utils.tcp_ip.serializer.DeSerializer method), 313

`bytes_to_int()` (py-modaq.utils.tcp_ip.serializer.DeSerializer static method), 313

`bytes_to_nd_array()` (py-modaq.utils.tcp_ip.serializer.DeSerializer static method), 313

`bytes_to_scalar()` (py-modaq.utils.tcp_ip.serializer.DeSerializer static method), 313

C

`channel_formatter()` (py-modaq.utils.h5modules.data_saving.DataToExportSaver static method), 284

`check_bound()` (pymodaq.control_modules.move_utility_classes.DAQ_Move_base method), 266

`check_received_length()` (py-modaq.utils.tcp_ip.mysocket.Socket method), 320

`check_received_length()` (py-modaq.utils.tcp_ip.serializer.SocketString method), 319

`check_sended()` (pymodaq.utils.tcp_ip.mysocket.Socket method), 320

`check_sended_with_serializer()` (py-modaq.utils.tcp_ip.mysocket.Socket method), 320

`check_version()` (py-modaq.utils.h5modules.browsing.H5Browser method), 290

`child_added()` (pymodaq.utils.managers.parameter_manager.ParameterManager method), 300

`close_file()` (pymodaq.utils.h5modules.backends.H5Backend method), 277

`close_server()` (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer method), 323

`cm2nm()` (in module `pymodaq.utils.units`), 326

`command` (pymodaq.utils.daq_utils.ThreadCommand attribute), 311

`commit_settings()` (py-modaq.control_modules.move_utility_classes.DAQ_Move_base method), 266

`connect_action()` (py-modaq.utils.managers.action_manager.ActionManager method), 298

`connect_actuators()` (py-modaq.utils.managers.modules_manager.ModulesManager method), 303

`connect_detectors()` (py-modaq.utils.managers.modules_manager.ModulesManager method), 304

`connect_things()` (py-modaq.extensions.BayesianOptimisation method), 269

`connect_things()` (py-modaq.utils.gui_utils.CustomApp method), 274

`Control Modules`, 261

`controller` (pymodaq.control_modules.move_utility_classes.DAQ_Move_base attribute), 264

`controller_units` (py-modaq.control_modules.move_utility_classes.DAQ_Move_base property), 268

`convert_input()` (py-modaq.extensions.BayesianModelDefault method), 272

`convert_input()` (py-modaq.extensions.BayesianModelGeneric method), 271

`convert_output()` (py-modaq.extensions.BayesianModelDefault method), 272

`convert_output()` (py-modaq.extensions.BayesianModelGeneric method), 271

`create_earray()` (py-modaq.utils.h5modules.backends.H5Backend method), 277

`create_linear_data()` (pymodaq.utils.data.Axis method), 332

`create_vlarray()` (py-modaq.utils.h5modules.backends.H5Backend method), 277

`crosshair` (pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D property), 307

`crosshair` (pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D property), 309

`current_value` (pymodaq.control_modules.move_utility_classes.DAQ_Move_base attribute), 264

`CustomApp` (class in `pymodaq.utils.gui_utils`), 273

D

`DAQ_Move_base` (class in `py-modaq.control_modules.move_utility_classes`), 264

`DashBoard`, 261

- `data` (`pymodaq.utils.data.Axis` property), 332
- `data` (`pymodaq.utils.data.DataBase` attribute), 334
- `data` (`pymodaq.utils.data.DataBase` property), 336
- `data` (`pymodaq.utils.data.DataToExport` attribute), 338
- `data` (`pymodaq.utils.data.DataToExport` property), 341
- `data_type` (`pymodaq.utils.h5modules.data_saving.AxisSaver` attribute), 282
- `data_type` (`pymodaq.utils.h5modules.data_saving.BkgSaver` attribute), 285
- `data_type` (`pymodaq.utils.h5modules.data_saving.DataEnlargeableSaver` attribute), 285
- `data_type` (`pymodaq.utils.h5modules.data_saving.DataExtendedSaver` attribute), 286
- `data_type` (`pymodaq.utils.h5modules.data_saving.DataManagement` attribute), 283
- `data_type` (`pymodaq.utils.h5modules.data_saving.DataSaverLoader` attribute), 283
- `DataBase` (class in `pymodaq.utils.data`), 333
- `DataCalculated` (class in `pymodaq.utils.data`), 337
- `DataDim`, 261
- `DataDim` (class in `pymodaq.utils.data`), 338
- `DataDistribution`, 262
- `DataDistribution` (class in `pymodaq.utils.data`), 338
- `DataEnlargeableSaver` (class in `py-modaq.utils.h5modules.data_saving`), 285
- `DataExtendedSaver` (class in `py-modaq.utils.h5modules.data_saving`), 285
- `DataFromPlugins` (class in `pymodaq.utils.data`), 337
- `DataFromRoi` (class in `pymodaq.utils.data`), 337
- `DataLoader` (class in `py-modaq.utils.h5modules.data_saving`), 288
- `DataManagement` (class in `py-modaq.utils.h5modules.data_saving`), 283
- `DataRaw` (class in `pymodaq.utils.data`), 337
- `DataSaverLoader` (class in `py-modaq.utils.h5modules.data_saving`), 283
- `DataSource`, 261
- `DataSource` (class in `pymodaq.utils.data`), 338
- `DataToExport` (class in `pymodaq.utils.data`), 338
- `DataToExportEnlargeableSaver` (class in `py-modaq.utils.h5modules.data_saving`), 286
- `DataToExportExtendedSaver` (class in `py-modaq.utils.h5modules.data_saving`), 287
- `DataToExportSaver` (class in `py-modaq.utils.h5modules.data_saving`), 284
- `DataToExportTimedSaver` (class in `py-modaq.utils.h5modules.data_saving`), 287
- `define_compression()` (`py-modaq.utils.h5modules.backends.H5Backend` method), 277
- `DeSerializer` (class in `pymodaq.utils.tcp_ip.serializer`), 312
- `Detector`, 261
- `DetectorEnlargeableSaver` (class in `py-modaq.utils.h5modules.module_saving`), 291
- `DetectorExtendedSaver` (class in `py-modaq.utils.h5modules.module_saving`), 291
- `Detectors` (`pymodaq.utils.managers.modules_manager.ModulesManager` property), 305
- `detectors_all` (`pymodaq.utils.managers.modules_manager.ModulesManager` property), 305
- `detect_h5_filename` (`pymodaq.utils.managers.modules_manager.ModulesManager` property), 305
- `DetectedSaver` (class in `py-modaq.utils.h5modules.module_saving`), 291
- `dict_from_param()` (in module `py-modaq.utils.parameter.ioxml`), 343
- `dim` (`pymodaq.utils.data.DataBase` attribute), 334
- `dim` (`pymodaq.utils.data.DataBase` property), 336
- `distribution` (`pymodaq.utils.data.DataBase` attribute), 334
- `distribution` (`pymodaq.utils.data.DataBase` property), 337
- `do_plot` (`pymodaq.utils.data.DataFromPlugins` attribute), 337
- `do_save` (`pymodaq.utils.data.DataFromPlugins` attribute), 337
- `dte`, 262
- `dte_deserialization()` (`py-modaq.utils.tcp_ip.serializer.DeSerializer` method), 313
- `dte_serialization()` (`py-modaq.utils.tcp_ip.serializer.Serializer` method), 315
- `dwa`, 262
- `dwa_deserialization()` (`py-modaq.utils.tcp_ip.serializer.DeSerializer` method), 313
- `dwa_serialization()` (`py-modaq.utils.tcp_ip.serializer.Serializer` method), 316
- E**
- `Ecmrel2Enm()` (in module `pymodaq.utils.units`), 325
- `elt_to_dict()` (in module `py-modaq.utils.parameter.ioxml`), 343
- `emit_new_file()` (`py-modaq.utils.h5modules.saving.H5Saver` method), 279
- `emit_status()` (`pymodaq.control_modules.move_utility_classes.DAQ_MoveUtilityClasses` method), 266
- `emit_value()` (`pymodaq.control_modules.move_utility_classes.DAQ_MoveUtilityClasses` method), 266
- `Enm2cmrel()` (in module `pymodaq.utils.units`), 325
- `eV2cm()` (in module `pymodaq.utils.units`), 326

eV2nm() (in module <i>pymodaq.utils.units</i>), 326	get_data_from_attribute() (py- <i>modaq.utils.data.DataToExport</i> method), 290	339
export_data() (<i>pymodaq.utils.h5modules.browsing.H5Browser</i> method), 291	get_data_from_dim() (py- <i>modaq.utils.data.DataToExport</i> method), 291	339
extra_attributes (<i>pymodaq.utils.data.DataBase</i> at- tribute), 335	get_data_from_dims() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
F	get_data_from_full_name() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
find_index() (in module <i>pymodaq.utils.math_utils</i>), 328	get_data_from_missing_attribute() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
find_index() (<i>pymodaq.utils.data.Axis</i> method), 332	get_data_from_name() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
find_part_in_path_and_subpath() (py- <i>modaq.utils.h5modules.saving.H5SaverBase</i> class method), 280	get_data_from_name_origin() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
find_socket_type_within_connected_clients() (py- <i>modaq.utils.tcp_ip.tcp_server_client.TCPServer</i> method), 323	get_data_from_Naxes() (py- <i>modaq.utils.data.DataToExport</i> method), 338	
find_socket_within_connected_clients() (py- <i>modaq.utils.tcp_ip.tcp_server_client.TCPServer</i> method), 324	get_data_from_sig_axes() (py- <i>modaq.utils.data.DataToExport</i> method), 339	
fliplr() (<i>pymodaq.utils.data.DataBase</i> method), 336	get_data_from_source() (py- <i>modaq.utils.data.DataToExport</i> method), 340	
flipud() (<i>pymodaq.utils.data.DataBase</i> method), 336	get_data_index() (<i>pymodaq.utils.data.DataBase</i> method), 336	
flush() (<i>pymodaq.utils.h5modules.module_saving.ModuleSaver</i> method), 292	get_data_with_naxes_lower_than() (py- <i>modaq.utils.data.DataToExport</i> method), 340	
ft() (in module <i>pymodaq.utils.math_utils</i>), 328	get_det_data_list() (py- <i>modaq.utils.managers.modules_manager.ModulesManager</i> method), 304	
ft2() (in module <i>pymodaq.utils.math_utils</i>), 328	get_dim_from_data() (<i>pymodaq.utils.data.DataBase</i> method), 336	
ftAxis() (in module <i>pymodaq.utils.math_utils</i>), 328	get_first_nbytes() (py- <i>modaq.utils.tcp_ip.mysocket.Socket</i> method), 320	
ftAxis_time() (in module <i>pymodaq.utils.math_utils</i>), 328	get_first_nbytes() (py- <i>modaq.utils.tcp_ip.serializer.SocketString</i> method), 319	
G	get_full_name() (<i>pymodaq.utils.data.DataBase</i> method), 336	
gauss1D() (in module <i>pymodaq.utils.math_utils</i>), 329	get_full_names() (<i>pymodaq.utils.data.DataToExport</i> method), 340	
gauss2D() (in module <i>pymodaq.utils.math_utils</i>), 329	get_h5_attributes() (py- <i>modaq.utils.h5modules.browsing.H5BrowserUtil</i> method), 291	
get_action() (<i>pymodaq.utils.managers.action_manager.ActionManager</i> method), 298	get_h5file_scans() (py- method), 309	
get_axes() (<i>pymodaq.utils.h5modules.data_saving.AxisSaverLoader</i> method), 282		
get_axes() (<i>pymodaq.utils.h5modules.data_saving.DataSaverLoader</i> method), 283		
get_axes_from_view() (py- <i>modaq.utils.plotting.data_viewers.viewer2D.Viewer2D</i> method), 308		
get_children() (<i>pymodaq.utils.h5modules.backends.H5Backend</i> method), 277		
get_data() (<i>pymodaq.utils.data.Axis</i> method), 332		
get_data() (<i>pymodaq.utils.tcp_ip.tcp_server_client.TCPServer</i> method), 322		
get_data_arrays() (py- <i>modaq.utils.h5modules.data_saving.DataSaverLoader</i> method), 283		
get_data_at() (<i>pymodaq.utils.data.Axis</i> method), 332		
get_data_at() (<i>pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D</i> method), 309		

`modaq.utils.h5modules.browsing.H5BrowserUtil` (method), 291
`get_indexes_from_scan_index()` (py-
`modaq.utils.scanner.Scanner` method), 294
`get_last_node()` (py-
`modaq.utils.h5modules.module_saving.ModuleSaver`
method), 292
`get_last_node_name()` (py-
`modaq.utils.h5modules.data_saving.DataManager`
method), 283
`get_last_scan()` (py-
`modaq.utils.h5modules.saving.H5SaverBase`
method), 280
`get_mod_from_name()` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
method), 304
`get_mods_from_names()` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
method), 304
`get_names()` (pymodaq.utils.data.DataToExport
method), 340
`get_names()` (pymodaq.utils.managers.modules_manager.ModulesManager
class method), 304
`get_nav_group()` (py-
`modaq.utils.h5modules.data_saving.DataLoader`
method), 288
`get_node()` (pymodaq.utils.h5modules.data_saving.DataLoader
method), 289
`get_node_name()` (py-
`modaq.utils.h5modules.backends.H5Backend`
method), 278
`get_node_path()` (py-
`modaq.utils.h5modules.backends.H5Backend`
method), 278
`get_origins()` (pymodaq.utils.data.DataToExport
method), 340
`get_param_from_name()` (in module py-
`modaq.utils.parameter.utils`), 345
`get_param_path()` (in module py-
`modaq.utils.parameter.utils`), 345
`get_position_with_scaling()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
method), 266
`get_scale_offset_from_data()` (py-
`modaq.utils.data.Axis` method), 332
`get_scan_index()` (py-
`modaq.utils.h5modules.saving.H5SaverBase`
method), 280
`get_scan_info()` (pymodaq.utils.scanner.Scanner
method), 294
`get_scanner_sub_settings()` (py-
`modaq.utils.scanner.Scanner` method), 294
`get_selected_probed_data()` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
method), 304
`get_set_group()` (py-
`modaq.utils.h5modules.backends.H5Backend`
method), 278
`get_set_node()` (pymodaq.utils.h5modules.module_saving.ModuleSaver
method), 292
`get_set_node()` (pymodaq.utils.h5modules.module_saving.ScanSaver
method), 293
`get_tree_node_path()` (py-
`modaq.utils.h5modules.browsing.H5Browser`
method), 290
`grab_data()` (pymodaq.utils.tcp_ip.tcp_server_client.Grabber
method), 321
`grab_datas()` (pymodaq.utils.managers.modules_manager.ModulesManager
class in py-
`modaq.utils.tcp_ip.tcp_server_client`), 321

H

`H5Backend` (class in py-
`modaq.utils.h5modules.backends`), 275
`H5Browser` (class in py-
`modaq.utils.h5modules.browsing`), 289
`H5BrowserUtil` (class in py-
`modaq.utils.h5modules.browsing`), 291
`H5Saver` (class in pymodaq.utils.h5modules.saving), 279
`H5SaverBase` (class in py-
`modaq.utils.h5modules.saving`), 279
`has_action()` (pymodaq.utils.managers.action_manager.ActionManager
method), 298

I

`ift()` (in module pymodaq.utils.math_utils), 330
`ift2()` (in module pymodaq.utils.math_utils), 330
`imag()` (pymodaq.utils.data.DataBase method), 336
`image_widget` (pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D
property), 309
`index` (pymodaq.utils.data.Axis property), 333
`index_from_name_origin()` (py-
`modaq.utils.data.DataToExport` method),
341
`ini_attributes()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
method), 267
`ini_model()` (pymodaq.extensions.BayesianModelDefault
method), 273
`ini_model()` (pymodaq.extensions.BayesianModelGeneric
method), 271
`ini_stage_init()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
method), 267
`init_file()` (pymodaq.utils.h5modules.saving.H5SaverBase
method), 280

[int_to_bytes\(\)](#) (`pymodaq.utils.tcp_ip.serializer.Serializer` static method), 317
[is_multiaxes](#) (`pymodaq.control_modules.move_utility_classes.Move_base` class attribute), 264
[is_node_in_group\(\)](#) (`pymodaq.utils.h5modules.backends.H5Backend` module method), 278
[isopen\(\)](#) (`pymodaq.utils.h5modules.data_saving.DataSaverLoader` module method), 284
[isopen\(\)](#) (`pymodaq.utils.h5modules.data_saving.DataToExportSaver` module method), 284
[ispolling](#) (`pymodaq.control_modules.move_utility_classes.DAQ_Move_base` property), 268
[iter_children\(\)](#) (in module `pymodaq.utils.parameter.utils`), 345
[iter_children_params\(\)](#) (in module `pymodaq.utils.parameter.utils`), 346
L
[l2w\(\)](#) (in module `pymodaq.utils.units`), 327
[label](#) (`pymodaq.utils.data.Axis` property), 333
[labels](#) (`pymodaq.utils.data.DataBase` attribute), 334
[LCD](#) (class in `pymodaq.utils.gui_utils.widgets.lcd`), 311
[length](#) (`pymodaq.utils.data.DataBase` attribute), 335
[length](#) (`pymodaq.utils.data.DataBase` property), 337
[linspace_step\(\)](#) (in module `pymodaq.utils.math_utils`), 330
[list_deserialization\(\)](#) (`pymodaq.utils.tcp_ip.serializer.DeSerializer` module method), 313
[list_serialization\(\)](#) (`pymodaq.utils.tcp_ip.serializer.Serializer` module method), 317
[listen_client\(\)](#) (`pymodaq.utils.tcp_ip.tcp_server_client.TCPServer` module method), 324
[load_axis\(\)](#) (`pymodaq.utils.h5modules.data_saving.AxisSaverLoader` module method), 283
[load_data\(\)](#) (`pymodaq.utils.h5modules.data_saving.DataBase` module method), 289
[load_data\(\)](#) (`pymodaq.utils.h5modules.data_saving.DataSaverLoader` module method), 284
[load_file\(\)](#) (`pymodaq.utils.h5modules.saving.H5SaverBase` module method), 281
[load_settings_slot\(\)](#) (`pymodaq.utils.managers.parameter_manager.ParameterManager` module method), 301
[LoggerSaver](#) (class in `pymodaq.utils.h5modules.module_saving`), 292
M
[menu](#) (`pymodaq.utils.managers.action_manager.ActionManager` property), 299
[merge_as_dwa\(\)](#) (`pymodaq.utils.data.DataToExport` module method), 341
[MockDAQClient](#) (class in `pymodaq.utils.tcp_ip.tcp_server_client`), 321
[Module](#), 262
[module](#)
[pymodaq.utils.daq_utils](#), 311
[pymodaq.utils.data](#), 332, 333, 338
[pymodaq.utils.gui_utils](#), 331
[pymodaq.utils.h5modules.browsing](#), 289
[pymodaq.utils.h5modules.data_saving](#), 282, 285
[pymodaq.utils.h5modules.module_saving](#), 291
[pymodaq.utils.h5modules.saving](#), 279
[pymodaq.utils.math_utils](#), 328
[pymodaq.utils.parameter.ioxml](#), 342
[pymodaq.utils.parameter.pymodaq_ptypes](#), 342
[pymodaq.utils.parameter.utils](#), 345
[pymodaq.utils.scanner](#), 331
[pymodaq.utils.tcp_ip.mysocket](#), 319
[pymodaq.utils.tcp_ip.serializer](#), 312
[pymodaq.utils.tcp_ip.tcp_server_client](#), 321
[pymodaq.utils.units](#), 325
[modules](#) (`pymodaq.utils.managers.modules_manager.ModulesManager` property), 306
[modules_all](#) (`pymodaq.utils.managers.modules_manager.ModulesManager` property), 306
[modules_manager](#) (`pymodaq.extensions.BayesianOptimisation` property), 270
[modules_manager](#) (`pymodaq.utils.gui_utils.CustomApp` property), 274
[ModuleSaver](#) (class in `pymodaq.utils.h5modules.module_saving`), 292
[ModulesManager](#) (class in `pymodaq.utils.managers.modules_manager`), 302
[move_actuators\(\)](#) (`pymodaq.utils.managers.modules_manager.ModulesManager` module method), 304
[move_done\(\)](#) (`pymodaq.control_modules.move_utility_classes.DAQ_Move_base` module method), 267
[move_done_signal](#) (`pymodaq.control_modules.move_utility_classes.DAQ_Move_base` attribute), 264
[move_roi_target\(\)](#) (`pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D` module method), 307
[move_roi_target\(\)](#) (`pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D` module method), 307

- method), 309
- my_moment() (in module pymodaq.utils.math_utils), 330
- ## N
- Nactuators (pymodaq.utils.managers.modules_manager.ModulesManager property), 305
- name (pymodaq.utils.data.DataBase attribute), 334
- name (pymodaq.utils.data.DataToExport attribute), 338
- Navigation, 262
- ndarray_deserialization() (pymodaq.utils.tcp_ip.serializer.DeSerializer method), 314
- ndarray_serialization() (pymodaq.utils.tcp_ip.serializer.Serializer method), 317
- Ndetectors (pymodaq.utils.managers.modules_manager.ModulesManager property), 305
- nm2cm() (in module pymodaq.utils.units), 327
- nm2eV() (in module pymodaq.utils.units), 327
- ## O
- object_type_serialization() (pymodaq.utils.tcp_ip.serializer.Serializer method), 318
- odd_even() (in module pymodaq.utils.math_utils), 330
- order_positions() (pymodaq.utils.managers.modules_manager.ModulesManager method), 305
- origin (pymodaq.utils.data.DataBase attribute), 334
- ## P
- param_deleted() (pymodaq.utils.managers.parameter_manager.ParameterManager method), 301
- parameter_to_xml_file() (in module pymodaq.utils.parameter.ioxml), 343
- parameter_to_xml_string() (in module pymodaq.utils.parameter.ioxml), 344
- ParameterManager (class in pymodaq.utils.managers.parameter_manager), 299
- params (pymodaq.control_modules.move_utility_classes.DAQ_Move_base attribute), 264
- params (pymodaq.utils.managers.parameter_manager.ParameterManager attribute), 299
- plot() (pymodaq.utils.data.DataToExport method), 341
- Plugin, 262
- poll_moving() (pymodaq.control_modules.move_utility_classes.DAQ_Move_base method), 267
- pop() (pymodaq.utils.data.DataBase method), 336
- pop() (pymodaq.utils.data.DataToExport method), 341
- populate_tree() (pymodaq.utils.h5modules.browsing.H5Browser method), 290
- positions_at() (pymodaq.utils.scanner.Scanner method), 294
- post_init() (pymodaq.utils.tcp_ip.tcp_server_client.TCPClient method), 322
- print_status() (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer method), 324
- process_cmds() (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer method), 324
- pymodaq.utils.daq_utils module, 311
- pymodaq.utils.data module, 332, 333, 338
- pymodaq.utils.gui_utils module, 331
- pymodaq.utils.h5modules.browsing module, 289
- pymodaq.utils.h5modules.data_saving module, 282, 285
- pymodaq.utils.h5modules.module_saving module, 291
- pymodaq.utils.h5modules.saving module, 279
- pymodaq.utils.math_utils module, 328
- pymodaq.utils.parameter.ioxml module, 342
- pymodaq.utils.parameter.pymodaq_ptypes module, 342
- pymodaq.utils.parameter.utils module, 345
- pymodaq.utils.scanner module, 331
- pymodaq.utils.tcp_ip.mysocket module, 319
- pymodaq.utils.tcp_ip.serializer module, 312
- pymodaq.utils.tcp_ip.tcp_server_client module, 321
- pymodaq.utils.units module, 325
- ## Q
- queue_command() (pymodaq.utils.tcp_ip.tcp_server_client.TCPClient method), 322
- quit_fun() (pymodaq.utils.h5modules.browsing.H5Browser method), 290
- ## R
- read_info() (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer

`method`), 324
`ready_to_read()` (py-
`modaq.utils.tcp_ip.tcp_server_client.TCPClient`
`method`), 322
`ready_to_write()` (py-
`modaq.utils.tcp_ip.tcp_server_client.TCPClient`
`method`), 322
`ready_with_error()` (py-
`modaq.utils.tcp_ip.tcp_server_client.TCPClient`
`method`), 322
`real()` (pymodaq.utils.data.DataBase `method`), 336
`roi_manager` (pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D
`property`), 307
`roi_manager` (pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D
`property`), 309
`roi_target` (pymodaq.utils.plotting.data_viewers.viewer1D.Viewer1D
`property`), 307
`roi_target` (pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D
`property`), 309
`runner_initialized()` (py-
`modaq.extensions.BayesianModelGeneric`
`method`), 272
S
`save_settings_slot()` (py-
`modaq.utils.managers.parameter_manager.ParameterManager`
`method`), 301
`scalar_deserialization()` (py-
`modaq.utils.tcp_ip.serializer.DeSerializer`
`method`), 314
`scalar_serialization()` (py-
`modaq.utils.tcp_ip.serializer.Serializer`
`method`), 318
`Scanner` (class in pymodaq.utils.scanner), 293
`ScanSaver` (class in py-
`modaq.utils.h5modules.module_saving`),
292
`ScanSelector` (class in py-
`modaq.utils.plotting.scan_selector`), 310
`select()` (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer
`method`), 324
`select_file()` (in module pymodaq.utils.gui_utils),
331
`selected_actuators_name` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
`property`), 306
`selected_detectors_name` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
`property`), 306
`send_command()` (pymodaq.utils.tcp_ip.tcp_server_client.TCPServer
`method`), 324
`send_param_status()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
`method`), 267
`Serializer` (class in pymodaq.utils.tcp_ip.serializer),
314
`set_action_text()` (py-
`modaq.utils.managers.action_manager.ActionManager`
`method`), 299
`set_actuators()` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
`method`), 305
`set_connected_clients_table()` (py-
`modaq.utils.tcp_ip.tcp_server_client.TCPServer`
`method`), 325
`set_crosshair_position()` (py-
`modaq.utils.plotting.data_viewers.viewer1D.Viewer1D`
`method`), 307
`set_crosshair_position()` (py-
`modaq.utils.plotting.data_viewers.viewer2D.Viewer2D`
`method`), 309
`set_current_scan_path()` (py-
`modaq.utils.h5modules.saving.H5SaverBase`
`class method`), 281
`set_detectors()` (py-
`modaq.utils.managers.modules_manager.ModulesManager`
`method`), 305
`set_dim()` (pymodaq.utils.data.DataBase `method`), 336
`set_gradient()` (pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D
`method`), 309
`set_image_transform()` (py-
`modaq.utils.plotting.data_viewers.viewer2D.Viewer2D`
`method`), 309
`set_menu()` (pymodaq.utils.managers.action_manager.ActionManager
`method`), 299
`set_position_relative_with_scaling()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
`method`), 267
`set_position_with_scaling()` (py-
`modaq.control_modules.move_utility_classes.DAQ_Move_base`
`method`), 267
`set_scan()` (pymodaq.utils.scanner.Scanner `method`),
294
`set_scan_type_and_subtypes()` (py-
`modaq.utils.scanner.Scanner` `method`), 295
`set_toolbar()` (pymodaq.utils.managers.action_manager.ActionManager
`method`), 299
`set_txt_from_elt()` (in module py-
`modaq.utils.parameter.ioxml`), 344
`settings` (pymodaq.control_modules.move_utility_classes.DAQ_Move_base
`attribute`), 264
`settings` (pymodaq.utils.h5modules.saving.H5SaverBase
`attribute`), 280
`settings` (pymodaq.utils.managers.parameter_manager.ParameterManager
`attribute`), 300
`settings_name` (pymodaq.utils.managers.parameter_manager.ParameterManager
`attribute`), 300
`settings_tree` (pymodaq.utils.h5modules.saving.H5SaverBase

- [attribute](#)), 280
 - [settings_tree](#) ([pymodaq.utils.managers.parameter_manager.ParameterManager](#) class attribute), 300
 - [setup_actions\(\)](#) ([pymodaq.extensions.BayesianOptimisation](#) method), 269
 - [setup_actions\(\)](#) ([pymodaq.utils.h5modules.browsing.H5Browser](#) method), 290
 - [setup_actions\(\)](#) ([pymodaq.utils.managers.action_manager.ActionManager](#) method), 299
 - [setup_actions\(\)](#) ([pymodaq.utils.plotting.data_viewers.viewerND.ViewerND](#) method), 309
 - [setup_docks\(\)](#) ([pymodaq.extensions.BayesianOptimisation](#) method), 270
 - [setup_docks\(\)](#) ([pymodaq.utils.gui_utils.CustomApp](#) method), 274
 - [setup_menu\(\)](#) ([pymodaq.extensions.BayesianOptimisation](#) method), 270
 - [setup_menu\(\)](#) ([pymodaq.utils.gui_utils.CustomApp](#) method), 274
 - [setvalues\(\)](#) ([pymodaq.utils.gui_utils.widgets.lcd.LCD](#) method), 311
 - [shape](#) ([pymodaq.utils.data.DataBase](#) attribute), 334
 - [shape](#) ([pymodaq.utils.data.DataBase](#) property), 337
 - [show_h5_data\(\)](#) ([pymodaq.utils.h5modules.browsing.H5Browser](#) method), 290
 - [show_roi\(\)](#) ([pymodaq.utils.plotting.data_viewers.viewer2D.Viewer2D](#) method), 309
 - [Signal](#), 262
 - [size](#) ([pymodaq.utils.data.Axis](#) property), 333
 - [size](#) ([pymodaq.utils.data.DataBase](#) attribute), 334
 - [size](#) ([pymodaq.utils.data.DataBase](#) property), 337
 - [Socket](#) (class in [pymodaq.utils.tcp_ip.mysocket](#)), 319
 - [SocketString](#) (class in [pymodaq.utils.tcp_ip.serializer](#)), 318
 - [source](#) ([pymodaq.utils.data.DataBase](#) attribute), 334
 - [source](#) ([pymodaq.utils.data.DataBase](#) property), 337
 - [stack_as_array\(\)](#) ([pymodaq.utils.data.DataBase](#) method), 336
 - [str_len_to_bytes\(\)](#) ([pymodaq.utils.tcp_ip.serializer.Serializer](#) class method), 318
 - [string_deserialization\(\)](#) ([pymodaq.utils.tcp_ip.serializer.DeSerializer](#) method), 314
 - [string_serialization\(\)](#) ([pymodaq.utils.tcp_ip.serializer.Serializer](#) method), 318
- ## T
- [target_value](#) ([pymodaq.control_modules.move_utility_classes.DAQ_Move_base](#) attribute), 265
 - [TCPClient](#) (class in [pymodaq.utils.tcp_ip.tcp_server_client](#)), 321
 - [TCPServer](#) (class in [pymodaq.utils.tcp_ip.tcp_server_client](#)), 322
 - [test_move_actuators\(\)](#) ([pymodaq.utils.managers.modules_manager.ModulesManager](#) method), 305
 - [ThreadCommand](#) (class in [pymodaq.utils.daq_utils](#)), 311
 - [timerEvent\(\)](#) ([pymodaq.utils.tcp_ip.tcp_server_client.TCPServer](#) method), 325
 - [timestamp](#) ([pymodaq.utils.data.DataToExport](#) attribute), 338
 - [to_bytes\(\)](#) ([pymodaq.utils.tcp_ip.serializer.Serializer](#) method), 318
 - [toolbar](#) ([pymodaq.utils.managers.action_manager.ActionManager](#) property), 299
 - [tree](#) ([pymodaq.utils.managers.parameter_manager.ParameterManager](#) attribute), 300
- ## U
- [units](#) ([pymodaq.utils.data.Axis](#) property), 333
 - [update_file_paths\(\)](#) ([pymodaq.utils.h5modules.saving.H5SaverBase](#) method), 281
 - [update_plots\(\)](#) ([pymodaq.extensions.BayesianModelGeneric](#) method), 272
 - [update_settings\(\)](#) ([pymodaq.control_modules.move_utility_classes.DAQ_Move_base](#) method), 267
 - [update_settings\(\)](#) ([pymodaq.extensions.BayesianModelDefault](#) method), 273
 - [update_settings\(\)](#) ([pymodaq.extensions.BayesianModelGeneric](#) method), 272
 - [update_settings_slot\(\)](#) ([pymodaq.utils.managers.parameter_manager.ParameterManager](#) method), 301
- ## V
- [value_changed\(\)](#) ([pymodaq.extensions.BayesianOptimisation](#) method), 270
 - [value_changed\(\)](#) ([pymodaq.utils.h5modules.saving.H5SaverBase](#) method), 281
 - [value_changed\(\)](#) ([pymodaq.utils.managers.modules_manager.ModulesManager](#) method), 305
 - [value_changed\(\)](#) ([pymodaq.utils.managers.parameter_manager.ParameterManager](#) method), 301

`value_changed()` (py-
modaq.utils.plotting.scan_selector.ScanSelector
method), 310

`value_changed()` (pymodaq.utils.scanner.Scanner
method), 295

`Viewer0D` (class in py-
modaq.utils.plotting.data_viewers.viewer0D),
306

`Viewer1D` (class in py-
modaq.utils.plotting.data_viewers.viewer1D),
306

`Viewer2D` (class in py-
modaq.utils.plotting.data_viewers.viewer2D),
308

`ViewerND` (class in py-
modaq.utils.plotting.data_viewers.viewerND),
309

W

`walk_nodes()` (pymodaq.utils.h5modules.data_saving.DataLoader
method), 289

`walk_parameters_to_xml()` (in module py-
modaq.utils.parameter.ioxml), 344

`walk_xml_to_parameter()` (in module py-
modaq.utils.parameter.ioxml), 345

X

`XML_file_to_parameter()` (in module py-
modaq.utils.parameter.ioxml), 342

`XML_string_to_parameter()` (in module py-
modaq.utils.parameter.ioxml), 343